AD-A045 102 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 9/2
SAIL,(U)
AUG 76 J F REISER MDA903-76-C-0206
UNCLASSIFIED STAN-CS-76-574 NL

1 OF 2
AD
A045 102

AD A045102

# SAIL

AUGUST 1976



DDC FILE COPY

DDC
RECEIVED
OCT 6 1977
A

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Stanford Artificial Intelligence Laboratory
Memo AIM-289

Computer Science Department
Report No. STAN-CS-76-574, AIM-289

August 1976

183p.

# SAIL

edited by

John F. Reiser

DDC
OCT 6 1977
A

ABSTRACT

Sail is a high-level programming language for the PDP-10 computer. It includes an extended ALGOL 60 compiler and a companion set of execution-time routines. In addition to ALGOL, the language features: (1) flexible linking to hand-coded machine language algorithms, (2) complete access to the PDP-10 I/O facilities, (3) a complete system of compile-time arithmetic and logic as well as a flexible macro system, (4) a high-level debugger, (5) records and references, (6) sets and lists, (7) an associative data structure, (8) independent processes (9) procedure varaiables, (10) user modifiable error handling, (11) backtracking, and (12) interrupt facilities.

This manual describes the Sail language and the execution-time routines for the typical Sail user: a non-novice programmer with some knowledge of ALGOL. It lies somewhere between being a tutorial and a reference manual.

This manual was supported by the Advanced Research Projects Agency under Contract MDA 903-76-C-0206.

The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the United States Government.

We thank Bernard A. Goldhirsch and the Institute for Advancement of Sailing for their kind permission to use the cover design of the August 1976 issue of SAIL magazine.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22161.



DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

# PREFACE

## HISTORY OF THE LANGUAGE

The GOGOL III compiler, developed principally by Dan Swinehart at the Stanford Artificial Intelligence Project, was the basis for the non-LEAP portions of SAIL. Robert Sproull joined Swinehart in incorporating the features of LEAP The first version of the language was released in November, 1969. SAIL's intermediate development was the responsibility of Russell Taylor, Jim Low, and Hanan Samet, who introduced processes, procedure variables, interrupts, contexts, matching procedures, a new macro system, and other features. Most recently John Reiser, Robert Smith, and Russell Taylor maintained and extended SAIL. They added a high-level debugger, conversion to TENEX, a print statement, and records and references.

## LEARNING ABOUT SAIL

A novice programmer (or one who is unfamiliar with ALGOL) should start with the Sail Tutorial [SmithN]. An experienced programmer with a knowledge of ALGOL should be able to use this Sail manual at once. Begin with Appendix A, Characters; in this manual the symbol "_" designates the character with code '030. For the first reading, a light skim of sections 1, 2, 3, 4, and 8, followed by a careful perusal of subsection 21.1 should be adequate to familiarize the new user with the differences between ALGOL and SAIL and allow him to start writing programs in SAIL. The other sections of this manual are relatively self contained, and can be read when one wants to know about the features they describe. The exceptions to this rule are sections 12, 13, and 14. These describe the basics of the LEAP and are essential for understanding of the following sections.

Special effort has gone into making the index more comprehensive than in previous versions of this manual. Please use it.

## CHANGES IN THE LANGUAGE

There are no known incompatibilities at the SAIL source level with the language described in [vanLehn]. PRINT, BAIL, operation under TENEX, and records are major additions to the language. Significant revisions to [vanLehn] or points deserving emphasis are marked by vertical bars in the margin. This paragraph is so marked, as an example.

## OPERATING SYSTEMS

Sail runs under several operating systems. In this manual distinction is drawn between the operating system at the Stanford Artificial Intelligence Laboratory (SUAI), the TOPS-10 operating system from Digital Equipment Corporation, the TENEX operating system from Bolt Beranek and Newman, and the TYMSHARE operating system. The major distinction is between TENEX and non-TENEX systems, although the differences between SUAI and TOPS-10 are also significant. The TOPS-20 operating system from Digital Equipment Corporation is the same as TENEX as far as Sail is concerned. TENEX users should substitute "<SAIL>" for "SYS:" wherever the latter appears in a file name (except when talking to the LOADER).

## UNIMPLEMENTED CONSTRUCTS

The following items are described in the manual as if they existed. As the manual goes to press, they are not implemented.

1. NE[illegible]text_variable>). Creates a new it[illegible] has a datum that is a context.
2. Using a <context_variable> instead of a list of variables in any of the REMEMBER, FORGET or RESTORE statements.
3. Using ∞ in the expression n of REMOVE n FROM list.
4. ANY⊗ANY≡ANY searches in Leap (searches where no constraints at all are placed on the triple returned.)
5. CHECKED itemvars (the dynamic comparison of the datum type of an item to the datum type of the CHECKED itemvar to which the item is being assigned.) It is currently the user's responsibility to insure that the type of the item agrees with the type of the itemvar whenever DATUM is used.

## ACKNOWLEDGEMENTS

Les Earnest and Robert Smith assisted the editor in PUB wizardry and reading drafts.

# TABLE OF CONTENTS


SECTION PAGE

1 PROGRAMS AND BLOCKS

1 Syntax 1
2 Semantics 1

2 ALGOL DECLARATIONS

1 Syntax 3
2 Restrictions 4
3 Examples 5
4 Semantics 5
5 Separately Compiled Procedures 12

3 ALGOL STATEMENTS

1 Syntax 14
2 Semantics 15

4 ALGOL EXPRESSIONS

1 Syntax 22
2 Type Conversion 23
3 Semantics 24

5 ASSEMBLY LANGUAGE STATEMENTS

1 Syntax 29
2 Semantics 29

6 INPUT/OUTPUT ROUTINES

1 Execution-time Routines in General 33
2 I/O Channels and Files 33
3 Break Characters 36
4 I/O Routines 39
5 TTY and PTY Routines 43
6 Example of TOPS-10 I/O 45

7 EXECUTION TIME ROUTINES

1 Type Conversion Routines 46
2 String Manipulation Routines 47
3 Liberation-from-Sail Routines 48
4 Byte Manipulation Routines 50
5 Other Useful Routines 50
6 Numerical Routines 51

8 PRINT

1 Syntax 53
2 Semantics 53

9 MACROS AND CONDITIONAL COMPILATION

1 Syntax 56
2 Delimiters 57
3 Macros 57
4 Macros with Parameters 59
5 Conditional Compilation 60
6 Type Determination at Compile Time 61
7 Miscellaneous Features 62
8 Hints 62

10 RECORD STRUCTURES

1 Introduction 64
2 Declaration Syntax 64
3 Declaration Semantics 64
4 Allocation 65
5 Fields 65
6 Garbage Collection 65
7 Internal Representations 66
8 Handler Procedures 66
9 More about Garbage Collection 67

11 TENEX ROUTINES

1 Introduction 69
2 TOPS-10 Style Input/Output 69
3 TENEX Style Input/Output 70
4 Terminal Handling 76
5 Utility TENEX System Calls 80

12 LEAP DATA TYPES

1 Introduction 83
2 Syntax 83
3 Semantics 84

13 LEAP STATEMENTS

1 Syntax 88
2 Restrictions 89
3 Semantics 89
4 Searching the Associative Store 91

14 LEAP EXPRESSIONS

1 Syntax 97
2 Semantics 98

15 BACKTRACKING

1 Introduction 101
2 Syntax 101
3 Semantics 101

16 PROCESSES

1 Introduction 104
2 Syntax 104
3 Semantics 104
4 Process Runtimes 107

17 EVENTS

1 Syntax 110
2 Introduction 110
3 Sail-defined Cause and Interrogate 110
4 User-defined Cause and Interrogate 112

18 PROCEDURE VARIABLES

1 Syntax 114
2 Semantics 114

19 INTERRUPTS

1 Introduction 117
2 Interrupt Routines 117
3 Immediate Interrupts 119
4 Clock Interrupts 120
5 Deferred Interrupts 121

20 LEAP RUNTIMES

1 Types and Type Conversion 123
2 Make and Erase Breakpoints 124
3 Pname Runtimes 124
4 Other Useful Runtimes 125
5 Runtimes for User Cause and Interrogate Procedures 126

21 BASIC CONSTRUCTS

1 Syntax 128
2 Semantics 128

22 USING SAIL

1 For TOPS-10 Beginners 131
2 For TENEX Beginners 131
3 The Complete use of Sail 132
4 Compiling Sail Programs 132
5 Loading Sail Programs 136
6 Starting Sail Programs 137
7 Storage Reallocation with REEnter 137

23 DEBUGGING SAIL PROGRAMS

1 Error Messages 138
2 Debugging 140
3 BAIL 141

APPENDICES

A Characters 150
B Sail Reserved Words 151
C Sail Predeclared Identifiers 152
D Indices for Interrupts 153
E Bit Names for Process Constructs 154
F Statement Counter System 156
G Array Implementation 157
H String Implementation 158
I Save/Continue 159
J Procedure Implementation 160

REFERENCES 163

ADDENDUM - February 1977 164a

INDEX 165

# SECTION 1

# PROGRAMS AND BLOCKS

## 1.1 Syntax

<program>
    ::= <block>

<block>
    ::= <block_head> ; <compound_tail>

<block_head>
    ::= BEGIN <declaration>
    ::= BEGIN <block_name> <declaration>
    ::= <block_head> ; <declaration>

<compound_tail>
    ::= <statement> END
    ::= <statement> END <block_name>
    ::= <statement> ; <compound_tail>

<compound_statement>
    ::= BEGIN <compound_tail>
    ::= BEGIN <block_name> <compound_tail>

<statement>
    ::= <block>
    ::= <compound_statement>
    ::= <require_specification>
    ::= <assignment>
    ::= <swap_statement>
    ::= <conditional_statement>
    ::= <if_statement>
    ::= <go_to_statement>
    ::= <for_statement>
    ::= <while_statement>
    ::= <do_statement>
    ::= <case_statement>
    ::= <print_statement>
    ::= <return_statement>
    ::= <done_statement>
    ::= <next_statement>
    ::= <continue_statement>
    ::= <procedure_statement>
    ::= <safety_statement>
    ::= <backtracking_statement>
    ::= <code_block>
    ::= <leap_statement>
    ::= <process_statement>
    ::= <event_statement>
    ::= <string_constant> <statement>
    ::= <label_identifier> : <statement>
    ::= <empty>

## 1.2 Semantics

DECLARATIONS

Sail programs are organized in the traditional block structure of ALGOL-60 [Nauer].

Declarations serve to define the data types and dimensions of simple and subscripted (array) variables (arithmetic variables, strings, sets, lists, record pointers, and items). They are also used to describe procedures (subroutines) and record classes, and to name program labels.

Any identifier referred to in a program must be described in some declaration. An identifier may only be referenced by statements within the scope (see page 5) of its declaration.

STATEMENTS

As in ALGOL, the statement is the fundamental unit of operation in the Sail language. Since a statement within a block or compound statement may itself be a block or compound statement, the concept of statement must be understood recursively.

The block representing the program is known as the "outer block". All blocks internal to this one will be referred to as "inner blocks".

BLOCK NAMES

The block name construct is used to describe the block structure of a Sail program to a symbolic debugging routine (see page 140). The name of the outer block becomes the title of the binary output file (not necessarily the file name). In addition, if a block name is used following an END then the compiler compares it with the block name which followed the corresponding BEGIN. A mismatch is reported to the user as evidence of a missing (extra) BEGIN or END somewhere.

The <string_constant> <statement> construct is equivalent in action to the <statement> alone; that is, the string constant serves only as a comment.

## EXAMPLES

Given:
S is a statement,
Sc is a Compound Statement,
D is a Declaration,
B is a Block.

Then:

(Sc) BEGIN S; S; S; ... ; S END

(Sc) BEGIN "SORT" S; S; ... ;S END "SORT"

(B) BEGIN D; D; D; ... ; S; S; S; ... ; S END

(B) BEGIN "ENTER NEW INFO" D; D; ... ;
S; ... ;S END

are syntactically valid Sail constructs.

# SECTION 2

# ALGOL DECLARATIONS

## 2.1 Syntax

<id_list>
    ::= <identifier>
    ::= <identifier> , <id_list>

<declaration>
    ::= <type_declaration>
    ::= <array_declaration>
    ::= <preload_specification>
    ::= <label_declaration>
    ::= <procedure_declaration>
    ::= <synonym_declaration>
    ::= <require_specification>
    ::= <context_declaration>
    ::= <leap_declaration>
    ::= <record_class_declaration>
    ::= <protect_acs declaration>
    ::= <cleanup_declaration>
    ::= <type_qualifier> <declaration>

<simple_type>
    ::= BOOLEAN
    ::= INTEGER
    ::= REAL
    ::= RECORD_POINTER ( <classid_list> )
    ::= STRING

<type_qualifier>
    ::= EXTERNAL
    ::= FORTRAN
    ::= FORWARD
    ::= INTERNAL
    ::= OWN
    ::= RECURSIVE
    ::= SAFE
    ::= SHORT
    ::= SIMPLE

<type_declaration>
    ::= <simple_type> <id_list>
    ::= <type_qualifier> <type_declaration>

<array_declaration>
    ::= <simple_type> ARRAY <array_list>
    ::= <type_qualifier> <array declaration>

<array_list>
    ::= <array_segment>
    ::= <array_list> , <array_segment>

<array_segment>
    ::= <id_list> [ <bound_pair_list> ]

<bound_pair_list>
    ::= <bound_pair>
    ::= <bound_pair_list> , <bound_pair>

<bound_pair>
    ::= <lower_bound> : <upper_bound>

<lower_bound>
    ::= <algebraic_expression>

<upper_bound>
    ::= <algebraic_expression>

<preload_specification>
    ::= PRELOAD_WITH <preload_list>
    ::= PRESET_WITH <preload_list>

<preload_list>
    ::= <preload_element>
    ::= <preload_list> , <preload_element>

<preload_element>
    ::= <expression>
    ::= [expression] <expression>

<label_declaration>
    ::= LABEL <id_list>

<procedure_declaration>
    ::= PROCEDURE <identifier>
        <procedure_head>
        <procedure_body>
    ::= <simple_type> PROCEDURE <identifier>
        <procedure_head> <procedure_body>
    ::= <type_qualifier>
        <procedure_declaration>

```
<procedure_head>
     ::= <empty>
     ::= ( <formal_param_decl> )

<procedure_body>
     ::= <empty>
     ::= ; <statement>

<formal_param_decl>
     ::= <formal_parameter_list>
     ::= <formal_parameter_list> ;
          <formal_param_decl>

<formal_parameter_list>
     ::= <formal_type> <id_list>
     ::= <formal_type> <id_list>
          ( <default_value> )

<formal_type>
     ::= <simple_formal_type>
     ::= REFERENCE <simple_formal_type>
     ::= VALUE <simple_formal_type>

<simple_formal_type>
     ::= <simple_type>
     ::= <simple_type> ARRAY
     ::= <simple_type> PROCEDURE

<synonym_declaration>
     ::= LET <synonym_list>

<synonym_list>
     ::= <synonym>
     ::= <synonym_list> , <synonym>

<synonym>
     ::= <identifier> = <reserved_word>

<cleanup_declaration>
     ::= CLEANUP <procedure_identifier_list>

<require_specification>
     ::= REQUIRE <require_list>

<require_list>
     ::= <require_element>
     ::= <require_list> , <require_element>

<require_element>
     ::= <constant_expression> <require_spec>
     ::= <procedure_name> INITIALIZATION
     ::= <procedure_name> INITIALIZATION
          [ <phase> ]

<require_spec>
     ::= STRING_SPACE
     ::= SYSTEM_PDL
     ::= STRING_PDL
     ::= ITEM_START
     ::= NEW_ITEMS
     ::= PNAMES
     ::= LOAD_MODULE
     ::= LIBRARY
     ::= SOURCE_FILE
     ::= SEGMENT_FILE
     ::= SEGMENT_NAME
     ::= POLLING_INTERVAL
     ::= POLLING_POINTS
     ::= VERSION
     ::= ERROR_MODES
     ::= DELIMITERS
     ::= NULL_DELIMITERS
     ::= REPLACE_DELIMITERS
     ::= UNSTACK_DELIMITERS
     ::= BUCKETS
     ::= MESSAGE
     ::= COMPILER_SWITCHES
```

## 2.2 Restrictions

For simplicity, the type_qualifiers are listed in only one syntactic class. Although their uses are always valid when placed according to the above syntax, most of them only have meaning when applied to particular subsets of these productions:

SAFE is only meaningful in array declarations.

INTERNAL/EXTERNAL have no meaning in formal parameter declarations.

SIMPLE, FORWARD, RECURSIVE, and FORTRAN have meaning only in procedure type specifications.

SHORT has meaning only when applied to INTEGER or REAL entities.

For array declarations in the outer block substitute <constant_expression> for <algebraic_expression> in the productions for <lower_bound> and <upper_bound>.

A label must be declared in the innermost block in which the statement being labeled appears (more information, page 16). The syntax for procedure declarations requires semantic embellishment (see page 7) in order to make total sense. In particular, a procedure body may be empty only in a restricted class of declarations.

## 2.3 Examples

Let I, J, K, L, X, Y, and P be identifiers, and let S be a statement.

```
(<type_declaration>)
        INTEGER I, J, K
        EXTERNAL REAL X, Y
        INTERNAL STRING K

(<array_declaration>)
        INTEGER ARRAY X [0:10, 0:10]
        REAL ARRAY Y [X:P(L)]; Comment illegal
             in outer block unless P is a macro
        STRING ARRAY I [0:IF BIG THEN 30 ELSE 3]

(<label_declaration>)
        LABEL L, X, Y

(<procedure declaration>)
        PROCEDURE P, S
        PROCEDURE P (INTEGER I, J;
             REFERENCE REAL X; REAL Y); S
        INTEGER PROCEDURE P (REAL PROCEDURE L;
             STRING I, J; INTEGER ARRAY K); S
        EXTERNAL PROCEDURE P (REAL X)
        FORWARD INTEGER PROCEDURE X (INTEGER I)
```

Note that these sample declarations are all given without the semicolons which would normally separate them from the surrounding declarations and statements. Here is a sample block to bring it all together (again, let S be any statement, D any declaration, and other identifiers as above):

```
BEGIN "SAMPLE BLOCK"
 INTEGER I, J, K;
 REAL X, Y;
 STRING A;
 INTEGER PROCEDURE P (REFERENCE REAL X);
  BEGIN "P"
   D; D; D; ... ;S; ... ; S
  END "P";

 REAL ARRAY DIPHTHONGS [0:10, 1:100];

 S; S; S; S
END "SAMPLE BLOCK"
```

## 2.4 Semantics

### SCOPE OF DECLARATIONS

Every block automatically introduces a new level of nomenclature. Any identifier declared in a block's head is said to be LOCAL to that block. This means that:

a. The entity represented by this identifier inside the block has no existence outside the block.

b. Any entity represented by the same identifier outside the block is completely inaccessible (unless it has been passed as a parameter) inside the block.

An identifier occurring within an inner block and not declared within that block will be nonlocal (global) to it; that is, the identifier will represent the same entity inside the block and in the block or blocks within which it is nested, up to and including the level in which the identifier is declared.

The Scope of an entity is the set of blocks in which the entity is represented, using the above rules, by its identifier. An entity may not be referenced by any statement outside its scope.

### TYPE QUALIFIERS

An array, variable, or procedure declared OWN will behave as if it were declared globally to the current procedure; the OWN type qualifier on a variable, etc. declared in a block not nested inside a procedure declaration will have no effect. This means that in a second call of a procedure with OWN locals (or a recursive call)

the OWN variables will not be reinitialized; they will have the values that they had when the first call of the procedure finished. Furthermore, OWN arrays, etc. will not be deallocated upon exiting the procedure in which they are declared.

INTERNAL and EXTERNAL procedures, variables, etc. let one link programs that are loaded together but were compiled separately. See page 12 for more information.

RECURSIVE, SHORT, FORTRAN, FORWARD, SIMPLE, and SAFE will be explained when the data types they modify are discussed.

NUMERIC DECLARATIONS

Identifiers which appear in type declarations with types REAL or INTEGER can subsequently be used to refer to numeric variables. An Integer variable may take on values from -2↑35 to 2↑35-1 (-2↑26 to 2↑26-1 for SHORT INTEGERS). A Real variable may take on positive and negative values from about 10↑-38 to 10↑38 with a precision of 27 bits (same range for SHORT REALs as for SHORT INTEGERs). REAL and INTEGER variables (and constants) may be used in the same arithmetic expressions; type conversions are carried out automatically (see page 23) when necessary.

The advantage of SHORT reals and integers is that the conversion from integer to real is sped by a factor of 8 if either the integer or the real is SHORT. See page 23 for more information.

The BOOLEAN type is identical to INTEGER. BOOLEAN and algebraic expressions are really equivalent syntactically. The syntactic context in which they appear determines their meaning. Non-zero integers correspond to TRUE and 0 corresponds to FALSE. The declarator BOOLEAN is included for program clarity.

STRING DECLARATIONS

A variable defined in a String declaration is a two-word descriptor containing the information necessary to represent a Sail character string.

A String may be thought of as a variable-length, one-dimensional array of 7-bit ASCII characters. Its descriptor contains a character count and a byte pointer to the first character (see page 158). Strings originate as constants at compile time (page 130), as the result of a String INPUT operation from some device (see page 39), or from the concatenation or decomposition of already existing strings (see page 27).

When strings appear in arithmetic operations or vice-versa, a somewhat arbitrary conversion is performed to obtain the proper type (by arbitrary we do not mean to imply random -- see page 23). For this reason arithmetic, String, and Record_pointer variables are referred to as "algebraic variables" and their corresponding expressions are called "algebraic expressions" (to differentiate them them from the variables and expressions of LEAP -- see page 83).

ARRAY DECLARATIONS

In general, any data type which is applicable to a simple variable may be applied in an Array declaration to an array of variables. The entity represented by the name of an Array, qualified with subscript expressions to locate a particular element (e.g. A[I, J]) behaves in every way like a simple variable. Therefore, in the future we shall refer to both simple variables and single elements of Arrays (subscripted variables) as "variables". The formal syntax for <variable> can be found on page 128.

For an Array which is not qualified by the SAFE attribute, nor had a NOW_SAFE statement done on it (Now_Safe - see page 21), each subscript will be checked to ensure that it falls within the lower and upper bounds given for the dimension it specifies. Subscripts outside the bounds trigger an error message and job abortion. The SAFE declaration inhibits this checking, resulting in faster, smaller, and bolder code.

Arrays which are allocated at compile time (OWN arrays and arrays in the outer block) are restricted to 5 or fewer dimensions. There is no limit to the number of dimensions allowed for an Array which is dynamically allocated. However, the efficiency of Array references tends to decrease for large dimensions. Avoid large dimensionality.

OWN Arrays are available in part. They must be declared with constant bounds, since fixed storage is allocated. They are NOT initialized when the program is started or restarted (except in preloaded Arrays, see page 7). A certain degree of extra efficiency is possible in accessing these Arrays, since they may be

assigned absolute core locations by the compiler, eliminating some of the address arithmetic. Constant bounds always add a little efficiency, even in inner blocks. Arrays declared in the outer block must have constant bounds, since no variable may yet have been assigned a value. They are thus automatically made OWN. For more details concerning the internal structure of Arrays see page 140 and page 157.

PRELOAD SPECIFICATIONS

Any OWN arithmetic or String Array may be "pre-loaded" at compile time with constant information by preceding its declaration with a <preload_specification>. This specification gives the values which are to be placed in consecutive core locations of the Arrays declared immediately following the <preload_specification>. "Immediately", in this case, means all identifiers up to and including one which is followed by bound_pair_list brackets (e.g. in REAL ARRAY X, Y, Z[0:10], W[1:5]; -- preloads X, Y, and Z; not W). It is the user's responsibility to guarantee that the proper values will be obtained under the subscript mapping, namely: arrays are stored by rows; if A[I, J] is stored in location 10000, then A[I, J+1] is stored in location 10001.

The current values of non-String pre-loaded Arrays will not be lost by restarting the program; they will not be re-initialized or re-preloaded. For preloaded String Arrays, the non-constant elements are set to NULL by a restart.

Algebraic type conversions will be performed at compile-time to provide values of the proper types to pre-loaded Arrays. All expressions in these specifications must be constant expressions -- that is, they must contain only constants and algebraic operators. The compiler will not allow you to fill an Array beyond its capacity. You may, however, provide a number of elements less than the total size of the Array; remaining elements will be set to zero or to the null string.

Example:

```
PRELOAD_WITH [5] 0, 3, 4, [4] 6, 2;
INTEGER ARRAY TABL [1:4, 1:3];
```

The first five elements of TABL will be initialized to 0 (bracketed number is used as a repeat argument). The next two elements will be 3 and 4, followed by four 6's and a 2. The array will look like this:

| (first subscript) | 1 | 2 | 3 (second subscript) |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 3 |
| 3 | 4 | 6 | 6 |
| 4 | 6 | 6 | 2 |

PRESET_WITH is just like PRELOAD_WITH except that an array which is PRESET is placed in the upper segment of a /H compilation. This allows constant arrays to be in the shared portion of the code.

PROCEDURE DECLARATIONS

If a Procedure is typed then it may return a value (see page 18) of the specified type. If formal parameters are specified then they must be supplied with actual parameters in a one to one correspondence when they are called (see page 28 and page 19).

FORMAL PARAMETERS

Formal parameters, when specified, provide information to the body (executable portion) of the Procedure about the kinds of values which will be provided as actual parameters in the call. The type and complexity (simple or Array) are specified here. In addition, the formal parameter indicates whether the value (VALUE) or address (REFERENCE) of the actual parameter will be supplied. If the address is supplied then the variable whose identifier is given as an actual parameter may be changed by the Procedure. This is not the case if the value is given.

To pass a PROCEDURE by value has no readily determined meaning. ARRAYs passed by value (requiring a complete copy operation) are not implemented. Therefore these cases are noted as errors by the compiler.

The proper use of actual parameters is further discussed on page 19 and page 28.

DEFAULT PARAMETER VALUES

Default values for trailing parameters may be specified by enclosing the desired value in parentheses following the parameter declaration.

```
PROCEDURE FOO (REAL X; INTEGER I (2);
    STRING S ("FOO"); REAL Y (3.14159) );
```

If a defaulted parameter is left out of a procedure call then the compiler fills in the default automatically. The following all compile the same code:

```
FOO (A+B);
FOO (A+B, 2, "FOO");
FOO (A+B, 2, "FOO", 3.14159);
```

Only VALUE parameters may be defaulted, and the default values must be constant expressions. A parameter may not be left out of the middle of the parameter list; i.e., FOO (A+B, , "BAR") won't work. Finally, it should be noted that the compiled code assumes that all parameters are actually present in the call, so be careful about odd START_CODE or INTERNAL-EXTERNAL linkages. However, APPLY will fill in default values if not enough actual parameters are supplied in an interpreted call.

## FORWARD PROCEDURE DECLARATIONS

A Procedure's type and parameters must be described before the Procedure may be called. Normally this is accomplished by specifying the procedure declaration in the head of some block containing the call. If, however, it is necessary to have two Procedures, declared in some block head, which are both accessible to statements in the compound tail of that block and to each other, then the FORWARD construct permits the definition of the parameter information for one of these Procedures in advance of its declaration. The Procedure body must be empty in a forward procedure declaration. When the body of the Procedure described in the forward declaration is actually declared, the types of the Procedure and of its parameters must be identical in both declarations. The declarations must appear at the same level (within the same block head). Example:

```
BEGIN "NEED FORWARD"
  FORWARD INTEGER PROCEDURE T1 (INTEGER I);
    COMMENT PARAMS DESCRIBED;
  INTEGER PROCEDURE T2 (INTEGER J);
    RETURN (T1 (J)+3); COMMENT CALL T1 ;
  INTEGER PROCEDURE T1 (INTEGER I);
    COMMENT ACTUALLY DEFINE T1;
    RETURN (IF I=15 THEN I ELSE T2 (I-1));
      COMMENT CALLS T2;
  ...
  K←T1 (L); ... ; L←T2 (K); ...
END "NEED FORWARD";
```

Notice that the forward declaration is required only because BOTH Procedures are called in the body of the block. These procedures should also be declared RECURSIVE if recursive entrance is likely. If only T1 were called from statements within the block then this example could be implemented as:

```
BEGIN "NO FORWARD"
  RECURSIVE INTEGER PROCEDURE T1 (INTEGER I);
  BEGIN
      INTEGER PROCEDURE T2 (J);
          RETURN (T1 (J)+3);
      RETURN( IF I=15 THEN I
                  ELSE T2 (I-1));
  END "T1";
  ...
  K←T1 (L);
...
END "NO FORWARD";
```

## RECURSIVE PROCEDURES

If a Procedure is to be entered recursively then the compiler must be instructed to provide code for allocating new local variables when the Procedure is called and deallocating them when it returns. Use the type-qualifier RECURSIVE in the declaration of any recursive Procedure.

The compiler can produce much more efficient code for non-recursive Procedures than for recursive ones. We feel that this gain in efficiency merits the necessity for declaring Procedures to be recursive.

If a Procedure which has not been declared recursive is called recursively then all its local variables (and temporary storage locations assigned by the compiler) will behave as if they were global to the Procedure -- they will not be reinitialized, and when the recursive call is complete, the locals of the calling procedure will reflect the changes made to them during the recursive call. Otherwise, no ill effects should be observed.

## SIMPLE PROCEDURES

Standard procedures contain a short prologue that sets up some links on the stack and a descriptor that is used by the storage allocation system, the GOTO solver, and some other routines. For most procedures, this overhead is insignificant. However, for small procedures that just do a few simple statements and exit, this overhead is excessive and unneeded. To

skip the prologue, just include SIMPLE in the attribute list for the procedure. RESTRICTIONS:

1. Simple procedures may not be Recursive and may not be SPROUTed or APPLYed.
2. ARRAY locals must be OWN.
3. Set and List locals must be OWN (Sets and list are part of Leap, page 83).
4. Procedures declared local to a simple procedure must also be of of type SIMPLE, and may not reference any of the parameters of the outer simple procedure.
5. One may not GO TO a statement outside the body of the simple procedure.
6. RECORD_POINTERs may not be declared or passed as arguments to other procedures, and the code must not cause the compiler to create *RECORD_POINTER temporaries.*

## EXTERNAL PROCEDURES

A file compiled by Sail represents either a "main" program or a collection of independent procedures to be called by the main program. The method for preparing such a collection of Procedures is described in page 12. The EXTERNAL and FORTRAN type-qualifiers allow description of the types of these Procedures and their parameters. An EXTERNAL or FORTRAN procedure declaration, like the FORWARD declaration, does not include a procedure body. Both declarations instead result in requests to the loader to provide the addresses of these Procedures to all statements which call them. This means that an EXTERNAL Procedure declaration (or the declaration of any External identifier) may be placed within any block head, thereby controlling the scope of this External identifier within this program.

Any Sail Procedure which is referenced via these external declarations must be an INTERNAL Procedure. That is, the type-qualifier INTERNAL must appear in the actual declaration of the Procedure. Again, see page 12.

The type-qualifier FORTRAN is used to describe the type and name of an external Procedure which is to be called using a Fortran calling sequence. Either the old F40 or the new FORTRAN-10 calling sequence can be generated, depending on the /A switch (page 134). All parameters to Fortran Procedures are by reference. In fact, the procedure head part of the declaration need not be included unless the types expected by the Procedure differ from those provided by the actual parameters--the number of parameters supplied, and their types, are presumed correct. Fortran Procedures are automatically External Procedures. See page 10, page 19, page 28 for more information about Fortran Procedures. Example:

```
FORTRAN PROCEDURE FPF;
Y←FPF (X, Z);
```

## PARAMETRIC PROCEDURES

The calling conventions for Procedures with Procedures as arguments, and for the execution of these parametric Procedures, are described on page 19 and page 28. Any Procedure PP which is to be used as a parameter to another Procedure CP must not have any Procedure or array parameters, or any parameters called by value. In other words, PP may only have simple reference parameters. The number of parameters supplied in a call on PP within CP, and their types, will be presumed correct, and should not be specified in the procedure head. Example:

```
PROCEDURE CP (INTEGER PROCEDURE FP);
   BEGIN INTEGER A, I; REAL X;
   ...
   A←FP (I, X); COMMENT I AND X PASSED BY
        REFERENCE, NO TYPE CONVERSION;
   END "CP";

INTEGER PROCEDURE PP (REFERENCE INTEGER J;
        REFERENCE REAL Y);
   BEGIN ...
   END "PP";
...
CP (PP);
```

## DEFAULTS IN PROCEDURE DECLARATIONS

If no VALUE or REFERENCE qualification appears in the description then the following qualifications are assumed:

| | |
|---|---|
| VALUE | Integer, String, Real, Record_pointer, Set, List variables |
| REFERENCE | Arrays, Contexts and Procedures |

## RESTRICTIONS ON PROCEDURE DECLARATIONS

1) Fortran Procedures cannot handle String parameters. Nor can a Fortran Procedure return a string as a result.

2) Labels may never be passed as arguments to Procedures.

3) Procedures may not have the type "CONTEXT".

4) Arrays and Context parameters must always be passed by reference.

## ALLOCATION AND DEALLOCATION

All simple variables (integer, real, string, boolean, record pointer) are allocated at compile time. Non-own simple variables that are local to a recursive procedure are an exception to this and are allocated (on the stack) upon instantiation of the procedure; they are deallocated when the instantiation is terminated. Simple variables which are declared but not subsequently referenced are not allocated at all.

All outer-block and OWN arrays are allocated at compile time. All other arrays are allocated when the block of their definition is entered, and deallocated when it is exited.

## INITIALIZATION AND REINITIALIZATION

Upon allocation, everything is initialized to 0 or the NULL string (except preloaded arrays, which are initialized to their the values of their PRELOAD). Nothing is reinitialized unless the program is restarted by typing ↑C and REEnter. This lack of reinitialization is noticeable when one enters a block for the second time, and that block is not the body of a recursive procedure. For example,

```
STRING PROCEDURE READIN;
BEGIN
   INTEGER CHANNEL, BRTAB;
   IF BRTAB=0 THEN BRTAB ← INIT (CHANNEL);
   RETURN (INPUT (CHANNEL, BRTAB));
END;
```

will return a string from an input operation with every call. However, on the first call, it will do some initialization of the I/O channel because BRTAB is 0 then, whereas it is not for any of the other calls. If READIN were a recursive procedure then CHANNEL and BRTAB would be allocated and hence initialized with every call.

When one REEnters a program, some things are reinitialized and some are not. Namely, strings and non-preloaded arrays will be reinitialized, but simple variables will not. Preloaded arrays will not be re-preloaded.

## SYNONYMS

The Sail Synonym ("LET") permits one to declare any identifier to act as a reserved word. The effect of the reserved word is not changed; it may be used as well as the new identifier. Synonyms follow the same scope rules that identifiers used for variables, arrays, etc. do.

Since Sail permits one to declare almost any reserved word to be an identifier for variables, procedures, etc. (see about restrictions on identifiers, page 129), synonyms are used to keep the effect of the reserved word available. For example,

```
LET BEG = BEGIN;
PROCEDURE BEGIN;
 BEG
   ....;

 END;

...
IF OK THEN BEGIN;
...
```

## CLEANUP DECLARATIONS

The CLEANUP declaration requires a list of procedure names following the "CLEANUP" token. Each procedure specified must be SIMPLE and have no formal parameters. The specified procedures will be called at the exit of the block that the CLEANUP declaration occurs in. They will be called in the order of their appearance on the list, and before any of the variables of the block are deallocated. NOTE: If the block is part of a process (see about processes, page 104) that is being terminated then the cleanup procedures will be called before the terminate is completed.

Cleanup procedures are normally used in connection with processes to "cleanup" a block by terminating the processes dependent on that

block (it is an error to leave active a process that depends on an exited block).

REQUIREMENTS

The user may, using the REQUIRE construct, specify to the compiler conditions which are required to be true of the execution-time environment of his programs. All requirements are legal at either declaration or statement level. The requirements fall into three classifications, described as follows:

Group 1 -- Space requirements -- STRING_SPACE, SYSTEM_PDL, etc.

The inclusion of the specification "REQUIRE 1000 STRING_SPACE" will ensure that at least 1000 words of storage will be available for storing (the text characters of) Strings when the program is run. Similar provisions are made for various push-down stacks used by the execution-time routines and the compiled code. If a parameter is specified twice, or if separately compiled procedures are loaded (see page 12) then the sum of all such specifications will be used. These parameters could also be typed to the loaded program just before execution (see page 137), but it is often more convenient to specify differences from the standard sizes in the source program. Use these specifications only if messages from the running program indicate that the standard allocations are not sufficient.

Group 2 -- Other files -- LOAD_MODULE, LIBRARY, SOURCE_FILE, etc.

The inclusion of the specification REQUIRE "PROCS1" LOAD_MODULE, "HELIB[1,3]" LIBRARY; would inform the Loader that the file PROCS1.REL must be loaded and the library HELIB.REL[1,3] searched whenever the program containing the specification is loaded. The parameter for both features should be a string constant of one of the above forms. The file extension .REL is the only value permitted, and is therefore assumed; the device, name, and ppn may be specified. TENEX users should note that the LOADER restricts LOAD_MODULE and LIBRARY file names to 6 characters in the main name and 3 characters in the extension.

LOAD_MODULES (.REL files to be loaded) may themselves contain requests for other LOAD_MODULES and LIBRARYs. LIBRARYs may only contain requests for other LIBRARYs. The LOADER may do strange things with files requested twice.

Sail automatically places a request for the library SYS:LIBSAn (<SAIL>LIBSAn on TENEX) [HLBSAn for /H compilations] in each main program, where n is the version number of the current Sail library of runtime routines.

The inclusion of REQUIRE "PREAMB.SAI" SOURCE_FILE will cause the compiler to save the state of the current input file, then begin scanning from PREAMB. When PREAMB is exhausted, Sail will resume scanning the original file on the line directly following the REQUIRE. Commonly-used declarations, particularly EXTERNAL declarations for libraries, are often put in a separate file which is then REQUIREd.

Restrictions: A SOURCE_FILE request must be followed by a semicolon (only one per REQUIREment), and must be the last text on the line in which it appears. SOURCE_FILE switching must not be specified from within a DEFINE body (see page 57). SOURCE_FILEs may be nested to a depth of about 10 levels.

The SEGMENT_NAME, SEGMENT_FILE specifications are currently applicable only to the SUAI "global model" users of Sail. They allow specification of the name of a special non-sharable "HISEG", and the name of the file used to create this HISEG. These specifications may, like the space REQUIREments, be overridden by using the system REENTER command (see page 137).

Group 3 -- other - INITIALIZATION, VERSION

Before the execution of a program, Sail runs through an initialization routine. The user can specify things that he wants done at initialization time by declaring an outer-block Procedure without arguments, then saying

REQUIRE procedure_name INITIALIZATION

Require-initialization procedures are run just before the first executable statement in the outer block of the program. They are run in order of ascending phase number, and within each phase in the order the compiler saw the REQUIREs. There are currently three user phases, numbered 0, 1, and 2. Phase 1 is the default if no phase is specified. WARNING: you should not Require initialization of a procedure which is declared inside another procedure.

REQUIRE n VERSION (n a non-zero integer) will flag the resultant .REL file as version n. When a program loaded from several such RELfiles is started, the Sail allocation code will verify that all specified versions are equal. A non-fatal error message is generated if any disagree. As much as will fit of the version number is also stored in lh(.JBVER), where .JBVER is location '137.

For other requirements, check the index under the specific condition being Required.

COMMENT: You have probably noticed that a great deal of prior knowledge is required for proper understanding of this section. For more information about storage allocation, see page 137 below. The form and use of .REL files and libraries are described in [TopHand].

## 2.5 Separately Compiled Procedures

When a program becomes extremely large it becomes useful to break it up into several files which can be compiled separately. This can be done in Sail by preparing one file as a main program, and one or more other files as programs each of which contains one or more procedures to be called by the main program. The main program must contain EXTERNAL declarations for each of the procedures declared in the other files. (EXTERNAL declarations have no procedure body.) The non-main program files must have the following characteristics:

1) All procedures to be called from the main program (or procedures in other files) must be qualified with the INTERNAL attribute when they are declared. External procedure declarations with headings identical to those of the actual declarations must appear in all those programs which call these procedures.

2) These internal procedures must be uniquely identifiable by the first six characters of their identifiers. In general, any two internal procedure names (or any other Internal variables in the same core image) with the same first six characters will cause incorrect linkages when the programs are loaded.

3) The reserved word ENTRY, followed by a semi-colon, must be the first item in the program (preceding even the BEGIN for its outer block). No starting address will be issued for a program containing an Entry Specification. Since no starting address is present for this file, entry to code within it may only be to the procedures it contains. The statements in the outer block, if any, can never be executed.

4) Should you desire your separatedly compiled procedures to be collected into a user library, include a list of their identifiers between the ENTRY and the semi-colon of the Entry Specification of the program containing those procedure declarations. The format of libraries is described in [TopHand]. The identifier(s) appearing in the entry list may be any valid identifiers, but usually they will be the names of the procedures contained in the file. No checking is done to see if entry identifiers are ever really declared in the body of the program.

5) Any variables (simple or array) which appear in the outer block of a Separately Compiled Procedure program will be global to the procedures in this program, but not available to the main program (unless they are themselves connected to the main program by Internal/External declarations -- see below). Non-LEAP arrays in these outer blocks will always be zero when the program is first loaded, but will never be cleared as others are by restarting your program (see reinitialization, page 10).

Any variable, procedure or label may contain the attribute INTERNAL or EXTERNAL in its declaration (ITEMS may not -- items are part of leap, page 83). The INTERNAL attribute does not affect the storage assignment of the entity it represents, nor does it have any effect on the behavior of the entity (or the scope of its identifier) in the file wherein it appears. However, its address and (the first six characters of) its name are made available to the loader for satisfying External requests. GOTO an external label is for wizards only.

No space is ever allocated for an External declaration. Instead, a list of references to each External identifier is made by the compiler. This list is passed to the loader along with the first six characters of the identifier name. (If there are no references then Sail ignores the External declaration.) When a matching Internal name is found during loading, the loader places the associated address in each of the instructions mentioned on the list. No program inefficiency at all results from External/Internal linkages (belay that -- references to External arrays are sometimes less efficient).

The entity finally represented by an External identifier is only accessible within the scope of the External declaration.

FORTRAN PROCEDURES

For a program written in either F40 or FORTRAN-10 to run in the Sail environment, the following restrictions must be observed:

1) It must be a SUBROUTINE or FUNCTION, not a main program.

2) It must not execute any FORTRAN I/O calls. The UUO structures of the two languages are not compatible.

3) It must be declared as a Fortran Procedure (see page 20) in the Sail program which calls it.

The type bits required in the argument addresses for Fortran arguments are passed correctly to these routines.

The Sail compiler will not produce a procedure to be called from FORTRAN.

ASSEMBLY LANGUAGE PROCEDURES

The following rules should be observed:

1) The ENTRY, INTERNAL, and EXTERNAL pseudo-ops should be used to obtain linkages for procedure names and "global" identifiers; remember that only six characters are used for these linkage names.

2) Accumulators F (currently '12), P (currently '17) and SP ('16) should be preserved over function calls. P may be used as a push-down pointer for arithmetic values and return addresses. SP is the string stack pointer. String results are returned on this stack. Arithmetic results are returned in AC 1.

3) Those who wish to provide their own UUO handlers or to increase their core size should read the code.

There are no other known processors which will produce Sail-compatible programs.

# SECTION 3

# ALGOL STATEMENTS

## 3.1 Syntax

```
<assignment_statement>
     ::= <algebraic_variable> ←
          <algebraic_expression>

<swap_statement>
     ::= <variable> ↔ <variable>
     ::= <variable> SWAP <variable>

<conditional_statement>
     ::= <if_statement>
     ::= <if_statement> ELSE <statement>

<if_statement>
     ::= IF <boolean_expression> THEN
          <statement>

<go_to_statement>
     ::= GO TO <label_identifier>
     ::= GOTO <label_identifier>
     ::= GO <label_identifier>

<label_identifier>
     ::= <identifier>

<for_statement>
     ::= FOR <algebraic_variable> ← <for_list>
          DO <statement>
     ::= NEEDNEXT <for_statement>

<for_list>
     ::= <for_list_element>
     ::= <for_list> , <for_list_element>

<for_list_element>
     ::= <algebraic_expression>
     ::= <algebraic_expression> STEP
          <algebraic_expression> UNTIL
          <algebraic_expression>
     ::= <algebraic_expression> STEP
          <algebraic_expression> WHILE
          <boolean_expression>

<while_statement>
     ::= WHILE <boolean_expression> DO
          <statement>
     ::= NEEDNEXT <while_statement>

<do_statement>
     ::= DO <statement> UNTIL
          <boolean_expression>

<case_statement>
     ::= <case_statement_head>
          <statement_list>
          <case_statement_tail>
     ::= <case_statement_head>
          <numbered_state_list>
          <case_statement_tail>

<case_statement_head>
     ::= CASE <algebraic_expression> OF BEGIN
     ::= CASE <algebraic_expression> OF BEGIN
          <block_name>

<case_statement_tail>
     ::= END
     ::= END <block_name>

<statement_list>
     ::= <statement>
     ::= <statement_list> ; <statement>

<numbered_state_list>
     ::= [ <integer_constant> ] <statement>
     ::= [ <integer_constant> ]
          <numbered_state_list>
     ::= <numbered_state_list> ;
          [ <integer_constant> ] <statement>

<return_statement>
     ::= RETURN
     ::= RETURN ( <expression> )

<done_statement>
     ::= DONE
     ::= DONE <block_name>

<next_statement>
     ::= NEXT
     ::= NEXT <block_name>
```

<continue_statement>
::= CONTINUE
::= CONTINUE <block_name>

<procedure_statement>
::= <procedure_call>

<procedure_call>
::= <procedure_identifier>
::= <procedure_identifier> (
<actual_parameter_list> )

<actual_parameter_list>
::= <actual_parameter>
::= <actual_parameter_list> ,
<actual_parameter>

<actual_parameter>
::= <expression>
::= <array_identifier>
::= <procedure_identifier>

<safety_statement>
::= NOW_SAFE <id_list>
::= NOW_UNSAFE <id_list>

## 3.2 Semantics

ASSIGNMENT STATEMENTS
The assignment statement causes the value represented by an expression to be assigned to the variable appearing to the left of the assignment symbol. You will see later (page 25) that one value may be assigned to two or more variables through the use of two or more assignment symbols. The operation of the assignment statement proceeds in the following order:

a) The subscript expressions of the left part variable (if any - Sail defines "variable" to include both array elements and simple variables) are evaluated from left to right (see Expression Evaluation Rules, page 25).

b) The expression is evaluated.

c) The value of the expression is assigned to the left part variable, with subscript expressions, if any, having values as determined in step a.

This ordering of operations may usually be disregarded. However it becomes important when expression assignments (page 25) or function calls with reference parameters appear anywhere in the statement. For example, in the statements:

```
K←3;
A[K]←3+(K←1);
```

A[3] will receive the value 4 using the above algorithm. A[1] will not change.

Any algebraic expression (REAL, INTEGER (BOOLEAN), or STRING) may be assigned to any variable of algebraic type. The resultant type will be that of the left part variable. The conversion rules for assignments involving mixed types are identical to the conversion rules for combining mixed types in algebraic expressions (see page 23).

SWAP ASSIGNMENT
The ↔ operator causes the value of the variable on the left hand side to be exchanged with the value of the variable on the right hand side. Arithmetic (REAL↔INTEGER) type conversions are made, if necessary; any other type conversions are invalid. Note that the ↔ operator may not be used in assignment expressions.

CONDITIONAL STATEMENTS
These statements provide a means whereby the execution of a statement, or a series of statements, is dependent on the logical value produced by a Boolean expression.

A Boolean expression is an algebraic expression whose use implies that it is to be tested as a logical (truth) value. If the value of the expression is 0 or NULL then the expression is a FALSE boolean expression, otherwise it is TRUE. See about type conversion, page 23.

IF STATEMENT - The statement following the operator THEN (the "THEN part") is executed if the logical value of the Boolean expression is TRUE; otherwise, that statement is ignored.

IF ... ELSE STATEMENT - If the Boolean expression is true, the "THEN part" is executed and the statement following the operator ELSE (the "ELSE part") is ignored. If the Boolean expression is FALSE, the "ELSE part" is executed and the "THEN part" is ignored.

### AMBIGUITY IN CONDITIONAL STATEMENTS

The syntax given here for conditional statements does not fully explain the correspondences between THEN-ELSE pairs when conditional statements are nested. An ELSE will be understood to match the immediately preceding unmatched THEN. Example:

```
COMMENT DECIDE WHETHER TO GO TO WORK;

IF ¬WEEKEND THEN
   IF GIANTS_ON_TV THEN BEGIN
      PHONE_EXCUSE ("GRANDMOTHER DIED");
      ENJOY (GAME);
      SUFFER (CONSCIENCE_PANGS)
   END
   ELSE IF REALLY_SICK THEN BEGIN
      PHONE_EXCUSE ("REALLY SICK");
      ENJOY (0);
      SUFFER (AGONY)
   END
   ELSE GO TO WORK;
```

### GO TO STATEMENTS

Each of the three forms of the Go To statement (GO, GOTO, GO TO) means the same thing -- an unconditional transfer is to be made to the "target" statement labeled by the label identifier. The following rules pertain to labels:

1) All label identifiers used in a program must be declared.

2) The declaration of a label must be local to the block immediately surrounding the statement it identifies (see exception below). Note that compound statements (BEGIN-END pairs containing no declarations) are not blocks. Therefore the block

```
BEGIN "B1"
 INTEGER I, J; LABEL L1;
 ...
 IF BE3 THEN BEGIN "C1"
  ...
  L1: ...
  ...
 END "C1";
 ...
 GO TO L1
END "B1"
```

is legal.

3) Rule 2 can be violated if the inner block(s) have no array declarations. E.g.:

| Legal | Illegal |
|---|---|
| `BEGIN "B1"` | `BEGIN "B1"` |
| `INTEGER I, J;` | `INTEGER I, J;` |
| `LABEL L1;` | `LABEL L1;` |
| `...` | `...` |
| `BEGIN "B2"` | `BEGIN "B2"` |
| `REAL X;` | `REAL ARRAY X [1:10];` |
| `...` | `...` |
| `L1: ...` | `L1: ...` |
| `...` | `...` |
| `END "B2";` | `END "B2";` |
| `GO TO L1;` | `GO TO L1;` |
| `END "B1"` | `END "B1"` |

4) No Go To statement may specify a transfer into a FOREACH statement (FOREACH statements are part of LEAP -- page 83), or into complicated For loops (those with For Lists or which contain a NEXT statement).

Labels will seldom be needed for debugging purposes. The block name feature (see page 140) and the listing feature which associates with each source line the octal address of its corresponding object code (see page 134) should provide enough information to find things easily.

Many program loops coded with labels can be alternatively expressed as For or While loops, augmented by DONE, NEXT, and CONTINUE statements. This often results in a source program whose organization is somewhat more transparent, and an object program which is more efficient.

## FOR STATEMENTS

For, Do and While statements provide methods for forming loops in a program. They allow the repetitive execution of a statement zero or more times. These statements will be described by means of Sail programs which are functionally equivalent but which demonstrate better the actual order of processing. Refer to these equations for any questions you might have about what gets evaluated when, and how many times each part is evaluated.

Let VBL be any algebraic variable, AE1, ... , AE8 any algebraic expressions, BE a Boolean expression, TEMP a temporary location, S a statement. Then the following Sail statements are equivalent.

Using For Statements:

```
FOR VBL ← AE1, AE2, AE3 STEP
   AE4 UNTIL AE5, AE6 STEP AE7 WHILE
   BE, AE8 DO S;
```

Equivalent formulation without For Statements:

```
VBL←AE1;
S;
VBL←AE2;
S;

VBL←AE3; Comment STEP-UNTIL loop;
LOOP1: IF (VBL-AE5) * SIGN(AE4) ≤ 0 THEN
   BEGIN
      S;
      VBL←VBL+AE4;
      GO TO LOOP1
   END;

VBL←AE6; Comment STEP-WHILE loop;
LOOP2: IF BE THEN BEGIN
      S;
      VBL←VBL+AE7;
      GO TO LOOP2
   END;

VBL←AE8;
S;
```

If AE4 (AE7) is an unsubscripted variable then changing its value within the loop will cause the new value to be used for the next iteration. If AE4 (AE7) is a constant or an expression requiring evaluation of some operator then the value used for the step element will remain constant throughout the execution of the For Statement. If AE5 is an expression then it will be evaluated before each iteration, so watch this possible source of inefficiency.

Now consider the For Statement:

```
FOR VBL←AE1 STEP CONST UNTIL AE2 DO S;
```

where const is a positive constant. The compiler will simplify this case to:

```
   VBL←AE1;
LOOP3: IF VBL ≤ AE2 THEN BEGIN
      S;
      VBL←VBL+CONST;
      GO TO LOOP3
   END;
```

If CONST is negative then the line at LOOP3 would be:

```
LOOP3: IF VBL ≥ AE2 THEN BEGIN
```

The value of VBL when execution of the loop is terminated, whether it be by exhaustion of the For list or by execution of a DONE, NEXT or GO TO statement (see page 18, page 19, page 16), is the value last assigned to it using the algorithm above. This value is therefore always well-defined.

The statement S may contain assignment statements or procedure calls which change the value of VBL. Such a statement behaves the same way it would if inserted at the corresponding point in the equivalent loop described above.

## WHILE STATEMENT

The statement:

```
WHILE BE DO S;
```

is equivalent to the statements:

```
LOOP: IF BE THEN BEGIN
      S;
      GO TO LOOP
   END;
```

## DO STATEMENT

The statement:

```
DO S UNTIL BE;
```

is equivalent to the sequence:

```
LOOP: S;
    IF ¬BE THEN GO TO LOOP;
```

## CASE STATEMENTS

The statement:

```
CASE AE OF BEGIN S0; S1; S2 ... Sn END
```

is functionally equivalent to the statements:

```
TEMP←AE;
IF TEMP<0 THEN ERROR
   ELSE IF TEMP = 0 THEN S0
   ELSE IF TEMP = 1 THEN S1
   ELSE IF TEMP = 2 THEN S2
   ...
   ELSE IF TEMP = n THEN Sn
   ELSE ERROR;
```

For applications of this type the CASE statement form will give significantly more efficient code than the equivalent *If statements.* Notice that dummy statements may be inserted for those cases which will not occur or for which no entries are necessary. For example,

```
CASE AE OF BEGIN S0; ; ; S3; ; ; S6; END
```

provides for no actions when AE is 1, 2, 4, 5, or 7. When AE is 0, 3, or 6 the corresponding statement will be executed. However, slightly more efficient code may be generated with a second type of Case statement that numbers each of its statement with [n] where n is an integer constant. The above example using this type of Case statement is then:

```
CASE AE OF BEGIN [3] S3; [0] S0; [6] S6 END;
```

All the statements must be numbered, and the numbers must all be non-negative integer constant expressions, although they may be in any order.

Multiple case numbers may precede each statement; the statement is executed for any one of the numbers specified. The following two CASE statements are equivalent:

```
CASE AE OF BEGIN [4] [1] S41; [2] [3] S23 END;
CASE AE OF BEGIN [1] S41; [2] S23;
   [3] S23; [4] S41 END;
```

Block names (i.e. any string constant) may be used after the BEGIN and END of a Case statement with the same effect as block names on blocks or compound statements. (See about block names on page 1).

## RETURN STATEMENT

This statement is invalid if it appears outside a procedure declaration. It provides for an early return from a Procedure execution to the statement calling the Procedure. If no return statement is executed then the Procedure will return after the last statement representing the procedure body is executed (see page 7).

An untyped Procedure (see page 19) may not return a value. The return statement for this kind of Procedure consists merely of the word RETURN. If an argument is given then it will cause the compiler to issue an error message.

A typed Procedure (see page 28) must return a value as it executes a return statement. If no argument is present an error message will be given. If the Procedure has an algebraic type then any algebraic expression may be returned as its value; type conversion will be performed in a manner described on page 23.

If no RETURN statement is executed in a typed Procedure then the value returned is undefined.

## DONE STATEMENT

The statement containing only the word DONE may be used to terminate the execution of a FOR, WHILE, or DO (also FOREACH - see page 92) loop explicitly. Its operation can most easily be seen by means of an example. The statement

```
FOR I←1 STEP 1 UNTIL n DO BEGIN
   S;
   ...
   IF BE THEN DONE;
   ...
END
```

is equivalent to the statement

```
FOR I←1 STEP 1 UNTIL n DO BEGIN
   S;
   ...
   IF BE THEN GO TO EXIT;
   ...
END;
EXIT:
```

In either case the value of I is well-defined after the statement has been executed (see page 17).

The DONE statement will only cause an escape from the innermost loop in which it appears, unless a block name follows "DONE". The block name must be the name of a block or compound statement (a "Loop Block") which is the object statement of some FOR, WHILE, or DO statement in which the current one is nested. The effect is to terminate all loops out to (and including) the Loop Block, continuing with the statement following this outermost loop. For example:

```
WHILE TRUE DO BEGIN "B1"
   ...
   IF OK THEN DO BEGIN "B2"
      ...
      FOR I←1 STEP 1 UNTIL K DO
         IF A[I]=FLAGWORD THEN DONE "B1";
      ...
      END "B2" UNTIL COWS_COME_HOME;
   ...
   END "B1";
```

Here the block named "B1" is the "loop block".

## NEXT STATEMENT

A Next statement is valid only in a For Statement or a While Statement (or Foreach - see page 92). Processing of the loop statement is temporarily suspended. When the NEXT statement appears in a For loop, the next value is obtained from the For List and assigned to the controlled variable. The termination test is then made. If the termination condition is satisfied then control is passed to the statement following the For Statement. If not, control is returned to the inner statement following the NEXT statement. In While and Do loops, the termination condition is tested. If it is satisfied, execution of the loop terminates. Otherwise it resumes at the statement within the loop following the NEXT statement.

Unless a block name follows NEXT, the innermost loop containing the NEXT statement is used as the "Loop Block" (see page 18). The terminating condition for the loop block is checked. If the condition is met then all inner loops are terminated (in DONE fashion) as well. If continuation is indicated then no inner-loop FOR-variable or WHILE-condition will have been affected by the NEXT code.

The reserved word NEEDNEXT must precede FOR or WHILE in the "Loop Block", and must not appear between this block and the NEXT statement. Example:

```
NEEDNEXT WHILE ¬EOF DO BEGIN
   S←INPUT(1,1);
   NEXT;
   Comment check EOF and terminate if TRUE;
   T←INPUT(1,3);
      PROCESS_INPUT(S,T);
   END;
```

## CONTINUE STATEMENT

The Continue statement is valid in only those contexts valid for the DONE statement (see page 18); the "Loop Block" is determined in the same way (i.e., implicitly or by specifying a block name). All loops out to the Loop Block are terminated as if DONE had been requested. Control is transferred to a point inside the loop containing the Loop Block, but after all statements in the loop. Example:

```
FOR I←1 STEP 1 UNTIL N DO BEGIN
     ...
     CONTINUE;
     ...
END
```

is semantically equivalent to:

```
FOR I←1 STEP 1 UNTIL N DO BEGIN
     LABEL CONT;
     ...
     GO TO CONT;
     ...
 CONT:
END
```

## PROCEDURE STATEMENTS

A Procedure statement is used to invoke the execution of a Procedure (see page 7). After execution of the Procedure, control returns to the statement immediately following the

Procedure statement. Sail does allow you to use typed Procedures as procedure statements. The value returned from the Procedure is simply discarded.

The actual parameters supplied to a Procedure must match the formal parameters described in the procedure declaration, modulo Sail type conversion. Thus one may supply an integer expression to a real formal, and type conversion will be performed as on page 23.

If an actual parameter is passed by VALUE then only the value of the expression is given to the Procedure. This value may be changed or examined by the Procedure, but this will in no way affect any of the variables used to evaluate the actual parameters. Any algebraic expression may be passed by value. Neither Arrays nor Procedures may be passed by value (use ARRBLT, page 51, to copy arrays). See the default declarations for parameters in page 9.

If an actual parameter is passed by REFERENCE then its address is passed to the Procedure. All accesses to the value of the parameter made by the Procedure are made indirectly through this address. Therefore any change the Procedure makes in a reference parameter will change the value of the variable which was used as an actual parameter. This is sometimes useful. However, if it is not intended, use of this feature can also be somewhat confusing as well as moderately inefficient. Reference parameters should be used only where needed.

Variables, constants, Procedures, Arrays, and most expressions may be passed by reference. No String expressions (or String constants) may be reference parameters.

If an expression is passed by reference then its value is first placed in a temporary location; a constant passed by reference is stored in a unique location. The address of this location is passed to the Procedure. Therefore, any values changed by the Procedure via reference parameters of this form will be inaccessible to the user after the Procedure call. If the called program is an assembly language routine which saves the parameter address, it is dangerous to pass expressions to it, since this address will be used by the compiler for other temporary purposes. A warning message will be printed when expressions are called by reference.

The type of each actual parameter passed by reference must match that of its corresponding formal parameter, modulo Sail type conversion. The exception is reference string formals, which must have string variables (or string array elements) passed to them. If an algebraic type mismatch occurs the compiler will create a temporary variable containing the converted value and pass the address of this temporary as the parameter, and a warning message will be printed. An exception is made for Fortran calls (see page 20).

### PROCEDURES AS ACTUAL PARAMETERS

If an actual parameter to a Procedure PC is the name of a Procedure PR with no arguments then one of three things might happen:

1) If the corresponding formal parameter requires a value of a type matching that of PR (in the loose sense given above in page 20), the Procedure is evaluated and its value is sent to the Procedure PC.

2) If the formal parameter of PC requires a reference Procedure of identical type, the address of PR is passed to PC as the actual parameter.

3) If the formal parameter requires a reference variable, the Procedure is evaluated, its result stored, and its address passed (as with expressions in the previous paragraph) as the parameter.

If a Procedure name followed by actual parameters appears as an actual parameter it is evaluated (see functions, page 28). Then if the corresponding formal parameter requires a value, the result of this evaluation is passed as the actual parameter. If the formal parameter requires a reference to a value, it is called as a reference expression.

### FORTRAN PROCEDURES

If the Procedure being called is a Fortran Procedure, all actual parameters must be of type INTEGER (BOOLEAN) or REAL. All such parameters are passed by reference, since Fortran will only accept that kind of call. For convenience, any constant or expression used as an actual parameter to a Fortran Procedure

is stored in a temporary cell whose address is given as the reference actual parameter.

It was explained in page 7 that formal parameters need not be described for Fortran Procedures. This allows a program to call a Fortran Procedure with varying numbers of arguments. No type conversion will be performed for such parameters, of course. If type conversion is desired, the formal parameter declarations should be included in the Fortran procedure declaration; Sail will use them if they are present.

To pass an Array to Fortran, mention the address of its first element (e.g. A[0], or B[1, 1]).

NOW_SAFE and NOW_UNSAFE

The NOW_SAFE and NOW_UNSAFE statements both take a list of Array names (names only - no indices) following them. From a NOW_SAFE until the end of the program or the next NOW_UNSAFE, the specified arrays will not have bounds checking code emitted for them. If an array has had a NOW_SAFE done on it, or has been declared SAFE, NOW_UNSAFE will cause bounds checking code to be emitted until the array is made safe again (if ever). Note that NOW_SAFE and NOW_UNSAFE are compile time statements. "IF BE THEN NOW_SAFE ..." will not work.

# SECTION 4

# ALGOL EXPRESSIONS

## 4.1 Syntax

```
<expression>
     ::= <simple_expression>
     ::= <conditional_expression>
     ::= <assignment_expression>
     ::= <case_expression>

<conditional_expression>
     ::= IF <boolean_expression> THEN
          <expression> ELSE <expression>

<assignment_expression>
     ::= <variable> ← <expression>

<case_expression>
     ::= CASE <algebraic_expression> OF (
          <expression_list> )

<expression_list>
     ::= <expression>
     ::= <expression_list> , <expression>

<simple_expression>
     ::= <algebraic_expression>
     ::= <leap_expression>

<boolean_expression>
     ::= <expression>

<algebraic_expression>
     ::= <disjunctive_expression>
     ::= <algebraic_expression> ∨
          <disjunctive_expression>
     ::= <algebraic_expression> OR
          <disjunctive_expression>

<disjunctive_expression>
     ::= <negated_expression>
     ::= <disjunctive_expression> ∧
          <negated_expression>
     ::= <disjunctive_expression> AND
          <negated_expression>

<negated_expression>
     ::= ¬ <relational_expression>
     ::= NOT <relational_expression>
     ::= <relational_expression>

<relational_expression>
     ::= <algebraic_relational>
     ::= <leap_relational>

<algebraic_relational>
     ::= <bounded_expression>
     ::= <relational_expression>
          <relational_operator>
          <bounded_expression>

<relational_operator>
     ::= <
     ::= >
     ::= =
     ::= ≤
     ::= ≥
     ::= ≠
     ::= LEQ
     ::= GEQ
     ::= NEQ

<bounded_expression>
     ::= <adding_expression>
     ::= <bounded_expression> MAX
          <adding_expression>
     ::= <bounded_expression> MIN
          <adding_expression>

<adding_expression>
     ::= <term>
     ::= <adding_expression> <add_operator>
          <term>

<adding_operator>
     ::= +
     ::= -
     ::= LAND
     ::= LOR
     ::= EQV
     ::= XOR

<term>
     ::= <factor>
     ::= <term> <mult_operator> <factor>
```

```
<mult_operator>
     ::= *
     ::= /
     ::= %
     ::= LSH
     ::= ASH
     ::= ROT
     ::= MOD
     ::= DIV
     ::= &

<factor>
     ::= <primary>
     ::= <primary> ↑ <primary>

<primary>
     ::= <algebraic_variable>
     ::= - <primary>
     ::= LNOT <primary>
     ::= ABS <primary>
     ::= <string_expression> [ <substring_spec>
         ]
     ::= ∞
     ::= INF
     ::= <constant>
     ::= <function_designator>
     ::= LOCATION ( <loc_specifier> )
     ::= ( <algebraic_expression> )

<string_expression>
     ::= <algebraic_expression>

<substring_spec>
     ::= <algebraic_expression> TO
         <algebraic_expression>
     ::= <algebraic_expression> FOR
         <algebraic_expression>

<function_designator>
     ::= <procedure_call>

<loc_specifier>
     ::= <variable>
     ::= <array_identifier>
     ::= <procedure_identifier>
     ::= <label_identifier>

<algebraic_variable>
     ::= <variable>
```

## 4.2 Type Conversion

Sail automatically converts between the data types Integer, Real, String and Boolean. The following table illustrates by description and example these conversions. The data type boolean is identical to integer under the mapping TRUE≠0 and FALSE=0.

| From \ To | INTEGER | REAL | STRING |
|---|---|---|---|
| INTEGER | | Left justify and raise to appropriate power.<br>1345→1.345@3<br>-678→-6.78@2 | Make a string of 1 character with the low 7 bits for its ASCII code.<br>48 → "0" |
| REAL | Take greatest integer.<br>1.345@2 → 134<br>-6.71@1 → -68<br>2.3@-2 → 0 | | Convert to integer, then to string.<br>4.8@1 → "0"<br>4.899@1 → "0" |
| STRING | The ASCII code for the first character of string.<br>"0SUM"→ 48<br>NULL → 0 | Convert to integer then to real.<br>"0SUM"→ 4.8@1<br>NULL → 0 | |

NOTES: The NULL string is converted to 0, but 0 is converted to the one character string with the ASCII code of 0. If an integer requires more than 27 bits of precision (2↑27 = 134217728) then some low order significance will be lost in the conversion to real; otherwise, conversion to real and then back to integer will result in the same integer value. If a real number has magnitude greater than 2↑35 - 2↑8 (=34359738112) then conversion to integer will produce an invalid result. UUOFIX does no error checking for this case; KIFIX and FIXR will set Overflow and Trap 1.

The default instruction compiled for a real to integer conversion is a UUO which computes FLOOR(x), the greatest integer function. This can be changed with the /A switch (page 134) to one of several other instructions. For real to integer conversion the choices are UUOFIX(opcode 003), KIFIX(122) and FIXR(126);

the effect of each is shown in the following table.

| real | UUOFIX | KIFIX | FIXR |
|---|---|---|---|
| 1.4 | 1 | 1 | 1 |
| 1.5 | 1 | 1 | 2 |
| 1.6 | 1 | 1 | 2 |
| -1.4 | -2 | -1 | -1 |
| -1.5 | -2 | -1 | -1 |
| -1.6 | -2 | -1 | -2 |

UUOFIX is the default. In mathematical terms, UUOFIX (x)=FLOOR (x)=[x] where [x] is the traditional notation for the greatest integer less than or equal to x. This UUO requires execution of 18.125 instructions (32 memory references) on the average. Many FORTRANs use the function implemented by KIFIX; KIFIX (x)=SIGN (x)*FLOOR (ABS (x)). Many ALGOLs use FIXR; FIXR (x)=FLOOR (x+0.5). Note that FIXR (-1.5) is not equal to -FIXR (1.5).

For integer to real conversion the choices are UUOFLOAT(002) and FLTR(127). FLTR rounds while UUOFLOAT (the default) truncates. It only makes a difference when the magnitude of the integer being converted is greater than 134217728. In such cases it is always true that UUOFLOAT (i)≤i and FLTR (i)≥i. UUOFLOAT merely truncates after normalization, while FLTR adds +0.5 lsb and then truncates. Most users will never see the difference. UUOFLOAT takes 18.625 instructions (32 memory references) on the average.

[For integer to real conversion involving a SHORT quantity, FSC ac,233 is used. At SUAI real to integer conversion involving a SHORT quantity uses KAFIX ac,233000; as this manual went to press KAFIX was simulated by the system and was very expensive.]

The binary arithmetic, logical, and String operations which follow will accept combinations of arguments of any algebraic types. The type of the result of such an operation is sometimes dependent on the type of its arguments and sometimes fixed. An argument may be converted to a different algebraic type before the operation is performed. The following table describes the results of the arithmetic and logical operations given various combinations of Real and Integer inputs. ARG1 and ARG2 represent the types of the actual arguments. ARG1' and ARG2' represent the types of the arguments after any necessary conversions have been made.

Beware: automatic type conversion can be a curse as well as a blessing. Study the conversion rules carefully; note that Sail has three division operators, %, DIV, and /.

| OPERATION | ARG1 | ARG2 | ARG1' | ARG2' | RESULT |
|---|---|---|---|---|---|
| + - | INT | INT | INT | INT | INT* |
| * ↑ % | REAL | INT | REAL | REAL | REAL |
| MAX MIN | INT | REAL | REAL | REAL | REAL |
| | REAL | REAL | REAL | REAL | REAL |
| LAND LOR | INT | INT | INT | INT | INT |
| EQV XOR | REAL | INT | REAL | INT | REAL |
| | INT | REAL | INT | REAL | INT |
| | REAL | REAL | REAL | REAL | REAL |
| LSH ROT | INT | INT | INT | INT | INT |
| ASH | REAL | INT | REAL | INT | REAL |
| | INT | REAL | INT | INT | INT |
| | REAL | REAL | REAL | INT | REAL |
| / | INT | INT | REAL | REAL | REAL |
| | REAL | INT | REAL | REAL | REAL |
| | INT | REAL | REAL | REAL | REAL |
| | REAL | REAL | REAL | REAL | REAL |
| MOD DIV | INT | INT | INT | INT | INT |
| | REAL | INT | INT | INT | INT |
| | INT | REAL | INT | INT | INT |
| | REAL | REAL | INT | INT | INT |

* For the operator ↑, ARG2' and RESULT are REAL unless ARG2 is a positive integer constant.

## 4.3 Semantics

### CONDITIONAL EXPRESSIONS

A conditional expression returns one of two possible values depending on the logical truth value of the Boolean expression. If the Boolean expression (BE) is true, the value of the conditional expression is the value of the expression following the delimiter THEN. If BE is false, the other value is used. If both expressions are of an algebraic type, the precise type of the entire conditional expression is that of the "THEN part". In particular, the "ELSE part" will be converted to the type of the "THEN part" before being returned as the value of the conditional expression. Reread and understand the last sentence.

Unlike the nested If statement problem, there can be no ambiguity for conditional expressions, since there is an ELSE part in every such expression. Example:

```
FOURTHDOWN (YARDSTOGO,YARDLINE,
    IF YARDLINE < 70 THEN PUNT ELSE
    IF YARDLINE < 90 THEN FIELDGOAL ELSE
    RUNFORIT)
```

## ASSIGNMENT EXPESSIONS

The somewhat weird syntax for an assignment expression (it is equivalent to that for an assignment statement) is nonetheless accurate: the two function identically as far as the new value of the left part variable is concerned. The difference is that the value of this left part variable is also retained as the value of the entire expression. Assuming that the assignment itself is legal (following the rules given in page 15 above), the type of the expression is that of the left part variable. This variable may now participate in any surrounding expressions as if it had been given its new value in a separate statement on the previous line. Only the ← operator is valid in assignment expressions. The ↔ operator is valid only at statement level. Example:

```
IF (K←K+1) < 30 THEN K←0 ELSE K←K+1;
```

## CASE EXPRESSIONS

*The expression*

```
CASE AE OF (E0, E1, E2, ... , En)
```

is equivalent to:

```
IF AE=0 THEN E0
   ELSE IF AE=1 THEN E1
   ELSE IF AE=2 THEN E2
   ...
   ELSE IF AE=n THEN En
   ELSE ERROR
```

The type of the entire expression is therefore that of E0. If any of the expressions E1 ... En cannot be fit into this mold an error message is issued by the compiler. Case expressions differ from Case statements in that one may not use the [n] construct to number the expressions. Example:

```
OUT (TTY, CASE ERRNO OF ("BAD DIRECTORY",
    "IMPROPER DATA MODE",
    "UNKNOWN I/O ERROR",
    ...
    "COMPUTER IN BAD MOOD"));
```

## SIMPLE EXPRESSIONS

Simple expressions are simple only in that they are not conditional, case, or assignment expressions. There are in fact some exciting complexities to be discussed with respect to simple expressions.

## PRECEDENCE OF ALGEBRAIC OPERATORS

The binary operators in Sail generally follow "normal" precedence rules. That is, exponentiations are performed before multiplications or divisions, which in turn are performed before additions and subtractions, etc. The bounding operators MAX and MIN are performed after these operations. The logical connectives ∧ and ∨, when they occur, are performed last (∧ before ∨). The order of operation can be changed by including parentheses at appropriate points.

In an expression where several operators of the same precedence occur at the same level, the operations are performed from left to right. See page 26 for special evaluation rules for logical connectives.

TABLE OF PRECEDENCE

```
↑
* / % & MOD DIV LSH ROT ASH
+ - ⊗ ≡ LAND LOR
MAX MIN
= ≠ < ≤ > ≥ LEQ GEQ NEQ
∧ AND
∨ OR
```

## EXPRESSION EVALUATION RULES

Sail does not evaluate expressions in a strictly left-to-right fashion. If we are not constrained to a left-to-right evaluation, (as is ALGOL 60), we can in some cases produce considerably better code than a strict left-to-right scheme could achieve. Intuitively, the essential features (and pitfalls) of this evaluation rule can be illustrated by a simple example:

```
b ← 26 ;
c ← b + (b ← b/2);
```

The second statement is executed as follows:

divide b by 2 and assign this value (1.3) to b. Add this value to b and assign the sum to c. Thus c gets 2.6. If the expressions were evaluated in a strictly left-to-right manner, c would get 2.6 + 1.3.

The evaluation scheme can be stated quite simply: code is generated for the operation represented by a BNF production when the reduction of that BNF production takes place. That is, b + (b ← b/2) isn't reduced until after (b ← b/2) is reduced, so the smaller expression gets done first.

"∨" (OR)
If an algebraic expression has as its major connective the logical connective "∨", the expression has the logical value TRUE (arithmetic value some non-zero integer) if either of its conjuncts (the expressions surrounding the "∨") is true; FALSE otherwise. The reserved word OR is equivalent to the symbol "∨". A∨B does NOT produce the bitwise Or of A and B if they are algebraic expressions. Truth values combined by numeric operators will in general be meaningless (use the operators LOR and LAND for bit operations).

The user should be warned that in an expression containing logical connectives, only enough of the expression is evaluated (from left to right) to uniquely determine its truth value. Thus in the expression

```
(J<3 ∨ (K←K+1) > 0),
```

K will not be incremented if J is less than 3 since the entire expression is already known to be true. Conversely in the expression

```
(X ≥0 ∧ SQRT(X)>2)
```

there is never any danger of attempting to extract the square root of a negative X, since the failure of the first test testifies to the falsity of the entire expression -- the SQRT routine is not even called in this case.

"∧" (AND)
If a disjunctive expression has as its major connective the logical connective "∧", the expression has the logical value TRUE if both of its disjuncts are TRUE; FALSE otherwise. Again, if the first disjunct is FALSE a logical value of FALSE is obtained for the entire expression without further evaluation. The reserved word AND is equivalent to "∧".

"¬" (NOT)
The unary Boolean operator ¬ applied to an argument BE (a relational expression, see Syntax) has the value TRUE if BE is false, and FALSE if BE is true. Notice that ¬A is not the bitwise complement of A, if A is an algebraic value. If used as an algebraic value, ¬A is simply 0 if A≠0 and some non-zero Integer otherwise. The reserved word NOT is equivalent to "¬".

"<>≤≥=≠" (RELATIONS)
If any of the binary relational operators is encountered, code is produced to convert any String arguments to Integer numbers. Then type conversion is done as it is for the + operations (see page 23). The values thus obtained are compared for the indicated condition. A Boolean value TRUE or FALSE is returned as the value of the expression. Of course, if this expression is used in subsequent arithmetic operations, a conversion to integer is performed to obtain an integer value. The reserved words LEQ, GEQ, NEQ are equivalent to "≤", "≥", "≠" respectively.

The syntax E1 RELOP1 E2 RELOP2 E3 where E1, E2, and E3 are expressions and RELOP1, RELOP2 are relational operators, is specially interpreted as (E1 RELOP1 (T←E2)) ∧ (T RELOP2 E3). The compiler can sometimes produce better code when the special syntax is used. Thus a bounds check may be written IF L<I<U THEN ... . RELOP1 and RELOP2 may be any relational operators, and need not be in transitive order. The following are equivalent:

```
IF A < X > B THEN ...        and
IF X > (A MAX B) THEN ...
```

MAX MIN
A MAX B (where A and B are appropriate expressions -- see the Syntax) has the value of the larger of A and B (in the algebraic sense). Type conversions are performed as if the operator were '+'. '0 MAX X MIN 10' is X if 0≤X≤10, 0 if X<0, 10 if X>10.

"+-" (ADDITION AND SUBTRACTION)
The + and - operators will do integer addition (subtraction) if both arguments are integers (or converted to integers from strings); otherwise, rounded Real addition or subtraction, after necessary conversions, is done.

LAND LOR XOR EQV LNOT

LAND, LOR, XOR, and EQV carry out bit-wise And, Or, Exclusive Or, and Equivalence operations on their arguments. No type conversions are done for these functions. The logical connectives ∧ and ∨ do not have this effect -- they simply cause tests and jumps to be compiled. The type of the result is that of the first operand. This allows expressions of the form X LAND '777777777, where X is Real, if they are really desired.

The unary operator LNOT produces the bitwise complement of its (algebraic) argument. No type conversions (except strings to integers) are performed on the argument. The type of the result (meaningful or not) is the type of the argument.

"*/%" (MULTIPLICATION AND DIVISION)

The operation * (multiplication), like + and -, represents Integer multiplication only if both arguments are integers; Real otherwise. Integer multiplication uses the IMUL machine instruction -- no double-length result is available.

The / operator (division) always does rounded Real division, after converting any Integer arguments to Real.

The % (division) operator has the same type table as +, -, and *. It performs whatever division is appropriate.

DIV MOD

DIV and MOD force both arguments to be integers before dividing. X MOD Y is the remainder after X DIV Y is performed:

X MOD Y = X - (X DIV Y)*Y .

ASH LSH ROT

LSH and ROT provide logical shift operations on their first arguments. If the value of the second argument is positive, a shift or rotation of that many bits to the left is performed. If it is negative, a right-shift or rotate is done. ASH does an arithmetic shift. Assume that A is an integer. If N is positive then the expression A ASH N is equal to A * 2↑N. If N is negative then A ASH N is equal to FLOOR (A / 2↑(-N)).

"&" (CONCATENATION)

This operator produces a result of type String. It is the String with length the sum of the lengths of its arguments, containing all the characters of the second string concatenated to the end of all the characters of the first. The operands will first be converted to strings if necessary as described in page 23 above. Numbers can be converted to strings representing their external forms (and vice-versa) through explicit calls on execution time routines like CVS and CVD (see page 46 below). NOTE: Concatenation of constant strings will be done at compile time where possible. For example, if SS is a string variable, SS&'12&'15 will result in two runtime concatenations, while SS&('12&'15) will result in one compile time concatenation and one runtime concatenation.

"↑" (EXPONENTIATION)

A factor is either a primary or a primary raised to a power represented by another primary. As usual, evaluation is from left to right, so that A↑B↑C is evaluated as (A↑B)↑C. In the factor X↑Y, a suitable number of multiplications and additions is performed to produce an "exact" answer if Y is a positive integer. Otherwise a routine is called to approximate ANTILOG (Y LOG X). The result has the type of X in the former case. It is always of type Real in the latter.

SUBSTRINGS

A String primary which is qualified by a substring specification represents a part of the specified string. The characters of a string STR are numbered 1, 2, 3, ..., LENGTH (STR). ST[X FOR Y] represents the substring which is Y characters long and begins with character X. ST[X TO Y] represents the Xth through Yth characters of ST.

Consider the ST[X TO Y] case. This is evaluated

```
_SKIP_←FALSE; XT←X; YT←Y;
IF YT > LENGTH (ST) THEN BEGIN
  YT←LENGTH (ST); righthalf (_SKIP_)←TRUE END;
IF YT < 0 THEN  COMMENT result will be NULL;
  BEGIN YT←0; righthalf (_SKIP_)←TRUE END;
IF XT < 1 THEN
  BEGIN XT ← 1; lefthalf (_SKIP_)←TRUE END;
IF XT > YT THEN  COMMENT result will be NULL;
  BEGIN XT ← YT+1; lefthalf (_SKIP_)←TRUE END;
<return the XTth through YTth characters of ST>
```

LENGTH returns the number of characters in a string (see page 48). The ST[X FOR Y] operation is converted to the ST[X TO Y] case before the substring operation is performed.

The variable _SKIP_ can be examined to determine if the substring indices were "out of bounds".

"∞" (SPECIAL LENGTH OPERATOR)
This special primary construct is valid only within substring brackets. It is an algebraic value representing the length of the most immediate string under consideration. The reserved word INF is equivalent to "∞". Example:

A[∞-2 to ∞] yields the last 3 characters of A.

A[3 for B[∞-1 for 1]] uses the next to the last character of string B as the number of characters for the A substring operation.

FUNCTION DESIGNATORS
A function designator defines a single value. This value is produced by the execution of a typed user Procedure or of a typed execution-time routine (See chapters 6 and 7 for execution-time routines). For a function designator to be an algebraic primary, its Procedure must be declared to have an algebraic type. Untyped Procedures may only be called as Procedure statements (see page 19). The value obtained from a user-defined Procedure is that provided by a Return Statement within that Procedure.

The rules for supplying actual parameters in a function designator are identical to those for supplying parameters in a procedure statement (see page 19).

UNARY OPERATORS
The unary operator ABS is valid only for algebraic quantities. It returns the absolute value of its argument.

-X is equivalent to (0-X). No type conversions are performed.

¬X is the logical negation of X.

MEMORY AND LOCATION
One's core image can be considered a giant one dimensional array, which may be accessed with the MEMORY construct. You had better be a good sport, or know what you are doing.

MEMORY [ <integer expression> ]

One can store and retrieve from the elements of MEMORY just as with any other array. However, with MEMORY, one can control how the compiler interprets the type of the accessed element by including type declarator reserved words after the <integer expression>. For example:

```
        ...← MEMORY[X, INTEGER]
MEMORY[X, REAL] ← ...
        ...← MEMORY[X, ITEM]
    COMMENT items and sets are part of Leap;
MEMORY[X, SET] ← ...
        ...← MEMORY[X, INTEGER ITEMVAR]
```

Note that one can not specify the contents of memory to be an Array or a String.

LOCATION is a predeclared Sail routine that returns the index in MEMORY of the Sail construct furnished it. The following is a list of constructs it can handle and what LOCATION will return.

| CONSTRUCT x | LOCATION (x) RETURNS |
|---|---|
| variable | address of the variable |
| string variable | -1,,address of word2 |
| array name | address of a word containing the the address of the first data word of the array |
| array element | address of that element |
| procedure name | address of the procedure's entry code |
| labels | address of the label |

Simple example:

```
REAL X;
MEMORY [LOCATION (X), REAL] ← 2.0;
PRINT (X); COMMENT " 2.000000   ";
MEMORY [LOCATION (X)] ← 2.0; PRINT (X);
  COMMENT " .0000000@-39", MEMORY is INTEGER
  unless otherwise specified;
MEMORY [LOCATION (X), INTEGER] ← 2.0;
PRINT (X); COMMENT same as above;
```

# SECTION 5

# ASSEMBLY LANGUAGE STATEMENTS

## 5.1 Syntax

```
<code_block>
     ::= <code_head> <code_tail>

<code_head>
     ::= <code_begin>
     ::= <code_begin> <block_name>
     ::= <code_head> <declaration> ;

<code_begin>
     ::= START_CODE
     ::= QUICK_CODE

<code_tail>
     ::= <instruction> END
     ::= <instruction> END <block_name>
     ::= <instruction> ; <code_tail>

<instruction>
     ::= <addresses>
     ::= <opcode>
     ::= <opcode> <addresses>

<addresses>
     ::= <address>
     ::= <ac_field> ,
     ::= <ac_field> , <address>

<ac_field>
     ::= <constant_expression>

<address>
     ::= <indexed_address>
     ::= @ <indexed_address>

<indexed_address>
     ::= <simple_address>
     ::= <simple_address> ( <index_field> )

<simple_address>
     ::= <identifier>
     ::= <static_array_name> [
          <constant_subscript_list> ]
     ::= <constant_expression>
     ::= <literal>

<literal>
     ::= [ <constant_expression> ]

<index_field>
     ::= <constant_expression>

<opcode>
     ::= <constant_expression>
     ::= <PDP-10_opcode>
```

## 5.2 Semantics

Within a START_CODE (QUICK_CODE) block, statements are processed by a small and weak, but hopefully adequate, assembly language translator. Each "instruction" places one instruction word into the output file. An instruction consists of

```
<label>:<opcode>    <ac_field>, @<simple_addr> (<index>)
```

or some subset thereof (see syntax). Each instruction must be followed by a semi-colon.

DECLARATIONS IN CODE BLOCKS

A code_block behaves like any other block with respect to block structure. Therefore, all declarations are valid, and the names given in these declarations will be available only to the instructions in the code_block. All labels must be declared as usual. Labels in code_blocks may refer to instructions which will be executed, or to those which are not really instructions, but data to be manipulated by these instructions (these latter words must be bypassed in the code by jump instructions). The user may find it easier to declare variables or SAFE arrays as data areas rather than using labels and null statements. As noted below, identifiers of simple variables are addresses of core locations. Identifiers of arrays are addresses of the first word of the array header (see the appendix on array implementation).

## PROTECT ACS DECLARATION

PROTECT_ACS <ac #>, ... , <ac #>;

where <ac #> is an integer constant between 0 and '17, is a declaration. Its effect is to cause Sail not to use the named accumulators in the code it emits for the block in which the declaration occurred (only AFTER the declaration). The most common use is with the ACCESS construct (see below); if one is using accumulators 2, 3, and 4 in a code block, then one should declare PROTECT_ACS 2, 3, 4 if one is going to use ACCESS. This way, the code emitted by Sail for doing the ACCESS will not use accumulators 2, 3, or 4. WARNING: this does not prevent you from clobbering such ACs with procedure calls (your own procedures or Sail's). However, most Sail runtimes save their ACs and restore them after the call.

RESTRICTION: Accumulators P ('17), SP ('16), F ('12) and 1 are used for, respectively, the system push down pointer, the string push down pointer, the display pointer, and returning results from typed procedures and runtimes. More about these acs on page 31. The protect mechanism will not override these usages, so attempts to protect 1, '12, '16, or '17 will be futile.

### OPCODES

The Opcode may be a constant provided by the user, or one of the standard (non I/O) PDP-10 operation codes, expressed symbolically. If a constant, it should take the form of a complete PDP-10 instruction, expressed in octal radix (e.g. DEFINE TTYUUO = "'51000000000";). Any bits appearing in fields other than the opcode field (first 9 bits) will be OR'ed with the bits supplied by other fields of instructions in which this opcode appears. In TOPS-10 Sail the MUUOs (ENTER, LOOKUP, etc.) are available. In TENEX Sail the JSYSes are available. Within a code_block opcodes supersede all other objects; a variable, macro, or procedure with the same name as an opcode will be taken for the opcode instead.

The indirect, index, and AC fields have the same syntax and perform the same functions as they do in the FAIL or MACRO languages.

### THE <simple addr> FIELD

If the <address> in an instruction is a constant (constant expression), it is assumed to be an immediate or data operand, and is not relocated. If the <address> is an identifier, the machine address (relative to the start of the compilation) is used, and will be relocated to the proper value by the Loader.

If the <address> is an identifier which has been declared as a formal parameter to a procedure, addressing arithmetic will be done automatically to get at the VALUE of the parameter. Hence if the <address> is a formal reference parameter, the instruction will be of the form OP AC,@ -x('12) where x depends on exactly where the parameter is in the stack. If the formal was from a simple procedure, then '17 will be used as the index register rather than '12. When computing x Sail will assume that the stack pointer has not changed since the last procedure entry; if you use PUSH, POP, etc. in a Simple Procedure then you must calculate x yourself.

If a literal is used, the address of the compiled constant will be placed in the instruction.

Any reference to Strings will result in the address of the second descriptor word (byte pointer) to be placed in the instruction (see the appendix on string implementation for an explanation of string descriptors).

Accessing parameters of procedures global to the current procedure is difficult. ACCESS (<expr>) may be used to return the address of such parameters. ACCESS will in fact do all of the computing necessary to obtain the value of the expression <expr>, then return the address of that value (which might be a temporary). Thus, MOVE AC, ACCESS(GP) will put the value of the variable GP in AC, while MOVEI AC, ACCESS(GP) will put the address of the variable GP in AC. If the expression is an item expression (see Leap), then the item's number will be stored in a temp, and that temp's address will be returned. The code emitted for an Access uses any acs that Sail believes are available, so one must include a PROTECT_ACS declaration in a Code block that uses ACCESS if you want to protect certain acs from being munged by the Access. WARNING: skipping over an Access won't do the right thing. For example,

```
SKIPE  FLAG;
MOVE   '10,ACCESS ('777 LAND INTIN(CHAN));
MOVEI  '10,0;
```

will cause the program to skip into the middle of the code generated by the access if FLAG is 0.

### START_CODE VERSUS QUICK_CODE

Before your instructions are parsed in a block starting with START_CODE, instructions are executed to leave all accumulators from 0 through '11 and '13 through '15 available for your use. In this case, you may use a JRST to transfer control out of the code_block, as long as you do not leave (1) a procedure, (2) a block with array declarations, (3) a Foreach loop, (4) a loop with a For list, or (5) a loop which uses the NEXT construct. In a QUICK_CODE block, no accumulator-saving instructions are issued. Ac's '13 through '15 only are free. In addition, some recently used variables may be given the wrong values if used as address identifiers (their current values may be contained in Ac's 0-'11); and control should not leave the code_block except by "falling through".

### ACCUMULATOR USAGE IN CODE BLOCKS

Although we have said that accumulators are "freed" for your use, this does not imply a *carte blanche*. Usually this means the compiler saves values currently stored in the ACs which it wants to remember (the values of variables mostly), and notes that when the code block is finished, these ACs will have values in them that it doesn't care about. However, this is not the case with the following accumulators, which are not touched at all by the entrance and exit of code blocks:

| NAME | NUMBER | USAGE |
|---|---|---|
| P | '17 | The system push down list pointer. All procedures are called with a PUSHJ P, PROC and exited (usually) with a POPJ P. Use this as your PDL pointer in the code block, but be sure that its back to where it was on entrance to the block by the time you exit. |
| SP | '16 | The string push down stack pointer. Used in all string operations. For how to do your own string mangling, read the code. |
| F | '12 | This is used to maintain the "display" structure of procedures. DO NOT HARM AC F!! Disaster will result. A more exact description of its usage may be found in the appendix on procedures and by reading the code. |

### CALLING PROCEDURES FROM INSIDE CODE BLOCKS

To call a procedure (say, PROT) from inside a code block, use PUSHJ P, PROT. If the procedure requires parameters, PUSH P them in order before you PUSHJ P (i.e. the first one first, the second one next, etc.). If the formal is a reference, push the address of the actual onto the P stack. If the formal is a value string, push onto the SP stack the two words of the string descriptor (see the appendix on string implementation for an explanation of string descriptors). If the formal is a reference string, simply PUSH P the address of the second word of the string descriptor. If the procedure is typed, it will return is value in AC 1, except that STRING procedures return their values as the top element of the SP stack. More information can be found in the appendix on procedure implementation. Example:

```
INTEGER K; STRING S, SS;
INTEGER PROCEDURE PROT (REAL T; REFERENCE
      INTEGER TT; STRING TTT; REFERENCE
      STRING TTTT);
            BEGIN COMMENT BODY; END;

DEFINE P = '17, SP = '16;

START_CODE
 PUSH    P, [3.14159];
 MOVEI   1, K;
 PUSH    P, 1;
 MOVEI   1, S;
 PUSH    SP, -1(1);      COMMENT if Sail allowed address
                         arithmetic in Start_code, you
                         could have said PUSH SP, S-1;
 PUSH    SP, S;
 MOVEI   1, SS;
 PUSH    P, 1;
 PUSHJ   P,PROT;
 END;
```

gives the same effect as

```
    PROT (3.14159, K, S, SS);
```

NOTE: procedures will change your

accumulators unless the procedure takes special pains to save and restore them.

BEWARE
The Sail <code block> assembler is not FAIL or MACRO. Read the syntax! Address arithmetic is not permitted. All integer constants are decimal unless specified explicitly as octal (e.g., '120). Each instruction is a separate <statement> and must be separated from surrounding statements by a semicolon. If you want comments then use COMMENT just like anywhere else in Sail. QUICK_CODE is for wizards.

# SECTION 6

# INPUT/OUTPUT ROUTINES

## 6.1 Execution-time Routines in General

SCOPE
A large set of predeclared, built-in procedures and functions have been compiled into a library permanently resident on the system disk area (SYS:LIBSAn.REL or <SAIL>LIBSAn.REL - n is the current version number; HLBSAn for /H compilations), and optionally into a special sharable write-protected high segment. The library also contains programs for managing storage allocation and initialization, and for certain String functions. If a user calls one of these procedures, a request is automatically made to the loader to include the procedure, and any other routines it might need, in the core image (or to link to the high segment). These routines provide input/output (I/O) facilities, Arithmetic-String conversion facilities, array-handling procedures and miscellaneous other interesting functions. The remainder of this section and the next describes the calling sequences and functions of these routines.

NOTATIONAL CONVENTIONS
A short-hand is used in these descriptions for specifying the types (if any) of the execution-time routines and of their parameters. Before the description of each routine there is a sample call of the form

```
VALUE ← FUNCTION (ARG1, ARG2, ... ARGn)
```

If VALUE is omitted, the procedure is an untyped one, and may only be called at statement level (page 19).

The types of VALUE and the arguments may be determined using the following scheme:

1) If " characters surround the sample identifier (which is usually mnemonic in nature) a String argument is expected. Otherwise the argument is Integer or Real. If it is important which of the types Integer or Real must be presented, it will be made clear in the description of the function. Otherwise the compiler assumes Integer arguments (for those functions which are predeclared). The user may pass Real arguments to these routines by re-declaring them in the blocks in which the Real arguments are desired.

2) If the @ character precedes the sample identifier, the argument will be called by reference. Otherwise it is a value parameter.

Example:

```
"RESULT" ← SCAN (@"SOURCE", BREAK_TABLE, @BRCHAR)
```

is a predeclared procedure with the implicit declaration:

```
EXTERNAL STRING PROCEDURE SCAN
    (REFERENCE STRING SOURCE;
     INTEGER BREAK_TABLE;
     REFERENCE INTEGER BRCHAR);
```

_SKIP_
Some routines return secondary values by storing them in _SKIP_. Declare EXTERNAL INTEGER _SKIP_ if you want to examine these values. In FAIL or DDT the spelling is ".SKIP.".

## 6.2 I/O Channels and Files

### OPEN

```
OPEN (CHANNEL, "DEVICE", MODE,
      NUMBER_OF_INPUT_BUFFERS,
      NUMBER_OF_OUTPUT_BUFFERS,
      @COUNT, @BRCHAR, @EOF);
```

Sail input/output operates at a very low level in the following sense: the operations necessary to obtain devices, open and close files, etc., are almost directly analogous to the system calls used in assembly language. OPEN is used to associate a channel number (0 to '17) with a device, to determine the data mode of the I/O to occur on this channel (character mode, binary mode, dump mode, etc.), to specify storage requirements for the data buffers used in the operations, and to provide the system with information to be used for

input operations. See page 45 for an example of TOPS-10 I/O programming.

CHANNEL is a user-provided channel number which will be used in subsequent I/O operations to identify the device. CHANNEL may range from 0 to 15 ('17). A RELEASE will be performed before the OPEN is executed.

DEVICE must be a String (i.e. "TTY", "DSK") which is recognizable by the system as a physical or logical device name.

MODE is the data mode for the I/O operation. MODE 0 will always work for characters (see INPUT, page 39 and OUT, page 40). Modes 8 ('10) and 15 ('17) are applicable for binary and dump-mode operations using the functions WORDIN, WORDOUT, ARRYIN, or ARRYOUT (see page 40 and following). For other data modes, see [SysCall]. If any of bits 18-21 are on in the MODE word, the I-O routines will not print error messages when data errors occur which present the corresponding bits as a response to the GETSTS UUO. Instead, the GETSTS bits will be reported to the user as described under EOF below. If bit 23 is on, no error message will be printed if an invalid file name specification is presented to LOOKUP, ENTER, or RENAME, a code identifying the problem will be returned (see page 36 ff. for details). If you don't understand any of this, leave all non-mode bits off in the MODE word.

NUMBER_OF_{INPUT/OUTPUT}_BUFFERS specifies the number of buffers to be reserved for the I/O operations. At least one buffer must be specified for input if any input is to be done in modes other than '17; similarly for output. If data is only going one direction, the other buffer specification should be 0. Two buffers give reasonable performance for most devices (1 is sufficient for a TTY, more are required for DSK if rapid operation is desired). The left half of the BUFFER parameter, if non-zero, specifies the buffer size (mod '7777) for the I/O buffers. Use this only if you desire non-standard sizes.

The remaining arguments are applicable only for INPUT (String input). They will be ignored for any other operations (although their values may be changed by the Open function).

COUNT designates a variable which will contain the maximum number of characters to be read from "DEVICE" in a given INPUT call (see page 39, page 36). Fewer characters may be read if a break character is encountered or if an end of file is detected. The count should be a variable or constant (not an expression), since its address is stored, and the temporary storage for an expression may be re-used.

BRCHAR designates a variable into which the break character (see INPUT and BREAKSET again) will be stored. This variable can be tested to determine which of many possible characters terminated the read operation.

EOF designates a variable to be used for two purposes:

1) Error handling when OPEN is called. If the system call used by OPEN succeeds then EOF is set to zero and OPEN returns. If the system call fails then OPEN looks at the EOF variable; if it is nonzero then OPEN returns. If EOF is zero then the user is given the option of retrying or continuing without the device. If a retry is successful then EOF is zeroed. If the user proceeds (gives up) then EOF is set to nonzero. The net effect is that the program may interpret EOF=0 as a successful OPEN and EOF≠0 as an unsuccessful OPEN.

2) Error handling for subsequent I/O operations. EOF will be made non-zero (TRUE) if an end of file condition, or any error condition among those enabled (see MODE, above) is detected during any Sail input/output operation. It will be 0 (FALSE) on return to the user otherwise. Subsequent inputs after an EOF return will return non-zero values in EOF and a null

String result for INPUT. For ARRYIN, a 0 is returned as the value of the call after end of file is detected. If EOF is TRUE after such an operation, it will contain the entire set (18 bits) of GETSTS information in the left half. The EOF bit is '20000, and is the only one you'll ever see if you haven't specially enabled for others.

Here are the error bits for SUAI and TOPS-10; TENEX Sail uses the ERSTR error number instead.

| | |
|---|---|
| 400000 | improper mode (a catchall) |
| 200000 | parity error |
| 100000 | data error |
| 40000 | record number out of bounds |
| 20000 | end of file (input only) |

You are always enabled for bit 20000 (EOF). However, to be allowed to handle any of the others, you must turn on the corresponding bit in the right half of the MODE word. In addition, the 10000 bit is used to enable user handling of invalid file specifications to ENTER, LOOKUP, and RENAME. '7500017 in the MODE parameter would enable a dump mode file for user handling of ALL I/O errors on the channel. If you are not enabled for a given error, an error message (which may or may not be fatal) will be printed, and the error code word set as indicated. In addition, the number of words actually transferred is stored in the right half of the EOF variable for ARRYIN, ARRYOUT.

Assembly Language Approximation to OPEN:

```
    INIT   CHANNEL, MODE
    SIXBIT /DEVICE/
    XWD    OHED,IHED
    JRST   <handle error condition>
    JUMPE  <NUMBER_OF_OUTPUT_BUFFERS>, GETIN
    <allocate buffer space>
    OUTBUF  CHANNEL, NUMBER_OF_OUTPUT_BUFFERS
GETIN: JUMPE  <NUMBER_OF_INPUT_BUFFERS>, DONE
    <allocate buffer space>
    INBUF  CHANNEL, NUMBER_OF_INPUT_BUFFERS
DONE:  <mark channel open -- internal bookkeeping>
    <return>

OHED: BLOCK 3
IHED: BLOCK 3
```

## CLOSE, CLOSIN, CLOSO

```
CLOSE (CHANNEL, BITS(0));
CLOSIN (CHANNEL, BITS(0));
CLOSO (CHANNEL, BITS(0))
```

The input (CLOSIN) or output (CLOSO) side of the specified channel is closed: all output is forced out (CLOSO); the current file name is forgotten. However the device is still active; no OPEN need be done again before the next LOOKUP/ENTER operation. Always CLOSE output files: Sail exit code will deassign the device, but does not force out any remaining output; you must do a CLOSE when writing on a disk file to have the new file (or a newly edited old file) entered on your User File Directory. No INPUT, OUT, etc., may be given to a directory device until an ENTER, LOOKUP, or RENAME has been issued for the channel.

CLOSE is equivalent to the execution of both CLOSIN and CLOSO for the channel. BITS specifies the close inhibit bits, which default to zero. See [SysCall] for the interpretation of the bits.

## GETCHAN

VALUE ← GETCHAN

GETCHAN returns the number of some channel which Sail believes is not currently open. The value -1 is returned if all channels are busy.

## RELEASE

```
RELEASE (CHANNEL, BITS(0))
```

If an OPEN has been executed for this channel, a CLOSE is now executed for it. The device is dissociated from the channel and returned to the resource pool (unless it has been assigned by the monitor ASSIGN command). No I/O operation may refer to this channel until another OPEN denoting it has been executed. BITS specifies the CLOSE inhibit bits; see [SysCall].

Release is always valid. If the channel mentioned is not currently open, the command is simply ignored.

### LOOKUP, ENTER

LOOKUP (CHANNEL, "FILE", @FLAG);
ENTER (CHANNEL, "FILE", @FLAG)

Before input or output operations may be performed for a directory device (DECtape or DSK) a file name must be associated with the channel on which the device has been opened (see page 33). LOOKUP names a file which is to be read. ENTER names a file which is to be created or extended (see [SysCall]). It is recommended that an ENTER be performed after every OPEN of an output device so that output not normally directed to the DSK can be directed there for later processing if desired. The format for a file name string is

"NAME", or
"NAME EXT", or
"NAME[P,PN]", or
"NAME EXT[P,PN]", or
"NAME EXT[P,PN"

See [MonCom] for the meaning of these things if you do not immediately understand.

Sail is not as choosy about the characters it allows as some processors are. Any character which is not a comma, period, right square bracket, or left square bracket will be passed on. Up to 6 characters from NAME, 3 from EXT, P, or PN will be used -- the rest are ignored.

If the LOOKUP or ENTER operation fails then variable FLAG may be examined to determine the cause. The left half of FLAG will be set to '777777 (Flag has the logical value TRUE). The right half will contain the code returned by the system giving the cause of the failure. An invalid file specification will return a code of '10. In this case, if the appropriate bit (bit 23, see OPEN) was OFF in the MODE parameter of the OPEN, an error message will be printed; otherwise, the routine just returns without performing the UUO.

If the LOOKUP or ENTER succeeds, FLAG will be set to zero (FALSE).

### RENAME

RENAME (CHANNEL, "FILE-SPEC",
PROTECTION, @FLAG)

The file open on CHANNEL is renamed to FILE_SPEC (a NULL file-name will delete the file) with read/write protection as specified in PROTECTION (nine bits, described in [SysCall]). FLAG is set as in LOOKUP and ENTER.

### ERENAME

ERENAME (CHANNEL,"FILE-SPEC",
PROTECTION, DATE, TIME,
MODE, @FLAG)

(Not on TENEX.) This extended version of RENAME allows complete specification of all the data which may be changed by a RENAME.

## 6.3 Break Characters

### BREAKSET

BREAKSET (TABLE, "BREAK_CHARS", MODE)

Character input/output is done using the String features of Sail. In fact, I/O is the chief justification for the existence of strings in the language.

String input presents a problem not present in String output. The length of an output String can be used to determine the number of characters written. However it is often awkward to require an absolute count for input. Quite often one would like to terminate input, or "break", when one of a specified set of characters is encountered in the input stream. In Sail, this capability is implemented by means of the BREAKSET, INPUT, TTYIN, and SCAN functions. The value of TABLE may range from -17 to 54, but tables -17 through -1 are reserved for use by the runtime system. Thus up to 54 different sets of user break specifications may exist at once. Which set will be used is determined by the TABLE parameter in an INPUT or SCAN function call. Breaktables

are dynamically allocated in blocks of 18 (1-18, 19-36, 37-54).

BREAKSET merely modifies the existing settings in TABLE; use GETBREAK (which returns a virgin table) if you want to achieve an absolute known state. The function of a given BREAKSET command depends on the MODE, an integer which is interpreted as a right-justified ASCII character whose value is intended to be vaguely mnemonic. BREAKSET commands can be partitioned into 4 groups according to mode:

GROUP 0 -- Conversion specifications

| MODE | FUNCTION |
|---|---|
| "K" | (Konvert) The minuscule letters (a-z) will be converted to majuscule (A-Z) before doing anything else. |
| "F" | (Full character set) Undoes the effect of "K". Mode "F" is the default. |
| "Z" | (Zero bytes) Believe the breaktable when INPUT reads a zero byte. INPUT automatically omits zero characters otherwise. Mode "Z" is turned off by both mode "I" and mode "X". |

GROUP 1 -- Break character specifications

| MODE | FUNCTION |
|---|---|
| "I" | (by Inclusion) The characters in the BREAK_CHARS String comprise the set of characters which will terminate an INPUT (or SCAN). |
| "X" | (by eXclusion) Only those characters (of the possible 128 ASCII characters) which are NOT contained in the String BREAK_CHARS will terminate an input when using this table. |
| "O" | (Omit) The characters in "BREAK_CHARS" will be omitted (deleted) from the input string. |

Any "I" or "X" command completely specifies the break character set for its table (i.e., the table is reset before these characters are stored in it). Neither will destroy the omitted character set currently specified for this table. Any "O" command completely specifies the set of omitted characters, without altering the break characters for the table in question. If a character is a break-character, any role it might play as an omitted character is sacrificed.

The next group of MODEs determines the disposition of break characters in the input stream. The "BREAK_CHARS" argument is ignored in these commands, and may in fact be NULL:

GROUP 2 -- Break character disposition

| MODE | FUNCTION |
|---|---|
| "S" | (Skip -- default mode) After execution of an "S" command the break character will not appear either in the resultant String or in subsequent INPUTs or SCANs-- the character is "skipped". Its value may be determined after the INPUT by examination of the break character variable (see page 33). |
| "A" | (Append) The break character (if there is one -- see page 33 and page 39) is appended, or concatenated to the end of the input string. It will not appear again in subsequent inputs. |
| "R" | (Retain) The break character does not appear in the resultant INPUT or SCAN String, but will be the first character processed in the next operation referring to this input source (file or SCAN String). |

Text files containing line numbers present a special problem. A line number is a word containing 5 ASCII characters representing the number in bits 0-34, with a "1" in bit 35. No other words in the file contain 1's in bit 35. Since String manipulations provide no way for distinguishing line numbers from other characters, there must be a way to warn the user that line numbers are present, or to allow him to ignore them entirely.

The next group of MODEs determines the disposition of these line numbers. Again, the "BREAK_CHARS" argument is ignored:

Group 3 -- Line number disposition

MODE FUNCTION

"P" (Pass -- default) Line numbers are treated as any other characters. Their identity is lost; they simply appear in the result string.

"N" (No numbers) No line number (or the TAB which always follows it in standard files) will appear in the result string. They are simply discarded.

"L" (Line no. break) The result String will be terminated early if a line number is encountered. The characters comprising the line number and the associated TAB will appear as the next 6 characters read or scanned from this character source. The user's break character variable (see page 33 and page 39) will be set to -1 to indicate a line number break.

"E" (Lee Erman's very own mode) The result String is terminated on a line number as with "L", but neither the line number nor the TAB following it will appear in subsequent inputs. The line number word, negated, is returned in the user's (integer) BRCHAR variable.

"D" (Display) obsolete

Once a break table is set up, it may be referenced in an INPUT, TTYIN or SCAN call to control the scanning operation.

Example: To delimit a "word", a program might wish to input characters until a blank, a TAB, a line feed, a comma, or a semicolon is encountered, ignoring line numbers. Assume also that carriage returns are to be ignored, and that the break character is to be retained in the character source for the next scanning operation:

```
BREAKSET (DELIMS, " , ;"&TAB&LF, "I");
   Comment break on any of these;

BREAKSET (DELIMS, '15, "O");
   Comment ignore carriage return;

BREAKSET (DELIMS, NULL, "N");
   Comment ignore line numbers;

BREAKSET (DELIMS, NULL, "R");
   Comment save break char for next time;
```

Breaktable 0 is builtin as equivalent to SETBREAK (0, NULL, NULL, "I"). This is break-on-count for INPUT and returns the whole string from SCAN.

## SETBREAK

```
SETBREAK (TABLE, "BREAK_CHARS",
          "OMIT_CHARS", "MODES")
```

SETBREAK is logically equivalent to the Sail statement:

```
BEGIN "SETBREAK"
 INTEGER I;

 IF LENGTH (OMIT_CHARS) > 0 THEN
   BREAKSET (TABLE, OMIT_CHARS, "O");

 FOR I←1 STEP 1 UNTIL LENGTH (MODES) DO
   BREAKSET (TABLE, BREAK_CHARS, MODES[I FOR 1])

END "SETBREAK"
```

## GETBREAK, RELBREAK

```
TABLE ← GETBREAK;
RELBREAK (TABLE)
```

GETBREAK finds an unreserved breaktable, reserves it, sets it to a completely virgin state, and returns the number of the table. GETBREAK returns -18 if there are no free tables. Breaktables are reserved by GETBREAK, SETBREAK, BREAKSET, and STDBRK. RELBREAK returns a table to the available list.

STDBRK

STDBRK (CHANNEL)

Eighteen breakset tables have been selected as representative of the more common input scanning operations. The function STDBRK initializes the breakset tables by opening the file SYS:BKTBL.BKT on CHANNEL and reading in these tables. The user may then reset those tables which he does not like to something he does like.

The eighteen tables are described here by giving the SETBREAKs which would be required for the user to initialize them:

```
DELIMS ← '15 & '12 & '40 & '11 & '14;
Comment carriage return, line feed, space,
        tab, form feed;
LETTS ← "ABC ... Zabc ... z_";
DIGS ← "0123456789";
SAILID ← LETTS&DIGS;

SETBREAK ( 1, '12, '15, "INS");
SETBREAK ( 2, '12, NULL, "INA");
SETBREAK ( 3, DELIMS, NULL, "XNR");
SETBREAK ( 4, SAILID, NULL, "INS");
SETBREAK ( 5, SAILID, NULL, "INR");
SETBREAK ( 6, LETTS, NULL, "XNR");
SETBREAK ( 7, DIGS, NULL, "XNR");
SETBREAK ( 8, DIGS, NULL, "INS");
SETBREAK ( 9, DIGS, NULL, "INR");
SETBREAK (10, DIGS&"+-@.", NULL, "XNR");
SETBREAK (11, DIGS&"+-@.", NULL, "INS");
SETBREAK (12, DIGS&"+-@.", NULL, "INR");
SETBREAK (13-18, NULL, NULL, NULL);
```

## 6.4 I/O Routines

INPUT

"RESULT" ← INPUT (CHANNEL, BREAK_TABLE)

A string of characters is obtained for the file open on CHANNEL, and is returned as the result. The INPUT operation is controlled by BREAK_TABLE (see page 36) and the reference variables BRCHAR, EOF, and COUNT which are provided by the user in the OPEN function for this channel (see page 33). Zero bytes are automatically omitted (text editor convention) unless mode "Z" was specified for the breaktable. Input may be terminated in several ways. The exact reason for termination can be obtained by examining BRCHAR and EOF:

| EOF | BRCHAR | |
|---|---|---|
| ≠0 | 0 | End of file or an error (if enabled, see page 33) occurred while reading. The result is a String containing all non-omitted characters which remained in the file when INPUT was called. |
| 0 | 0 | No break characters were encountered. The result is a String of length equal to the current COUNT specifications for the CHANNEL (see page 33). |
| 0 | <0 | A line number was encountered and the break table specified that someone wanted to know. The result String contains all characters up to the line number. If mode "L" was specified in the Breakset setting up this table, bit 35 is turned off in the line number word so that it will be input next time. -1 is placed in BRCHAR. If mode "E" was specified, the line number will not appear in the next input String, but its negated ASCII value, complete with low-order line number bit, will be found in BRCHAR. |
| 0 | >0 | A break character was encountered. The break character is stored in BRCHAR (an INTEGER reference variable, see page 33) as a right-justified 7-bit ASCII value. It may also be tacked on to the end of the result String or saved for next time, depending on the BREAKSET mode (see page 36). |

If break table 0 is specified, the only criteria for termination are end of file or COUNT exhaustion.

## SCAN

"RESULT" ← SCAN (@"SOURCE",
BREAK_TABLE, @BRCHAR)

SCAN functions identically to INPUT with the following exceptions:

1. The source is not a data file but the String SOURCE, called by reference. The String SOURCE is truncated from the left to produce the same effect as one would obtain if SOURCE were a data file. The disposition of the break character is the same as it is for INPUT.

2. BRCHAR is directly specified as a parameter. INPUT gets its break character variable from a table set up by page 33.

3. Line number considerations are irrelevant.

## SCANC

"RESULT" ← SCANC ("SOURCE",
"BREAK", "OMIT", "MODE");

This routine is equivalent to the following Sail code:

```
STRING PROCEDURE SCANC (STRING ARG, BRK,
      OMIT, MODE);
BEGIN "SCANC" INTEGER TBL, BRCHAR; STRING RSLT;
TBL←GETBREAK; SETBREAK (TBL, BRK, OMIT, MODE);
RSLT←SCAN (ARG, TBL, BRCHAR);
RELBREAK (TBL);
RETURN (RSLT) END "SCANC";
```

Note that the arguments are all value parameters, so that SCANC will be called at compile time if the arguments are constants. It is intended that SCANC be used with ASSIGNC in macros and conditional compilation. For scanning at execution time, it is much more efficient to use SCAN directly.

## OUT

OUT (CHANNEL, "STRING")

STRING is output to the file open on CHANNEL. If the device is a TTY, the String will be typed immediately. Buffered mode text output is employed for this operation. The data mode specified in the OPEN for this channel must be 0 or 1. The EOF variable will be set non-zero as described in page 33 if an error is detected and the program is enabled for it; 0 otherwise.

## LINOUT

LINOUT (CHANNEL, NUMBER)

ABS (NUMBER) mod 100,000 is converted to a 5 character ASCII string. These characters are placed in a single word in the output file designated by CHANNEL with the low-order bit (line-number bit) turned on. A tab is inserted after the line number. Mode 0 or 1 must have been specified in the OPEN (page 33) for the results to be anywhere near satisfactory. EOF is set as in OUT.

## SETPL

SETPL (CHANNEL, @LINNUM,
@PAGNUM, @SOSNUM)

This routine allows one to keep track of the string input from CHANNEL. Whenever a '12 is encountered, LINNUM is incremented. Whenever a '14 is encountered, PAGNUM is incremented and LINNUM is zeroed. Whenever an SOS line number is encountered it is placed into SOSNUM.

## WORDIN

VALUE ← WORDIN (CHANNEL)

The next word from the file open on CHANNEL is returned. A zero is returned, and EOF (see page 33, page 39) set, when end of file or error is encountered. This operation is performed in buffered mode or dump mode, depending on the mode specification in the OPEN.

WARNING ABOUT DUMP MODE IO
Dump Mode (mode '15, '16, or '17) is sufficiently device and system dependent that you should consult [SysCall] and be extremely careful.

## ARRYIN

ARRYIN (CHANNEL, @LOC, HOW_MANY)

HOW_MANY words are read from the device and file open on CHANNEL, and deposited in memory starting at location LOC. Buffered-mode input is done if MODE (see page 33) is '10 or '14. Dump-mode input is done if MODE is '16 or '17. Other modes are illegal. See the warning about Dump Mode IO above. If an end of file or enabled error condition occurs before HOW_MANY words are read in buffered mode then the EOF variable (see page 33) is set to the enabled bits in its left half, as usual. Its right half contains the number of words actually read. EOF will be 0 if the full request is satisfied. No indication of how many words were actually read is given if EOF is encountered while reading a file in DUMP mode.

## WORDOUT

WORDOUT (CHANNEL, VALUE)

VALUE is placed in the output buffer for CHANNEL. An OUTPUT is done when the buffer is full or when a CLOSE or RELEASE is executed for this channel. Dump mode output will be done if dump mode is specified in the OPEN (see page 33). EOF is set as in OUT. See the warning about Dump Mode IO above.

## ARRYOUT

ARRYOUT (CHANNEL, @LOC, HOW_MANY)

HOW_MANY words are written from memory, starting at location LOC, onto the device and file open on channel CHANNEL. The valid modes are again '10, '14, '16, and '17. The EOF variable is set as in ARRYIN, except that the EOF bit itself will never occur.

## INOUT

INOUT (INCHAN, OUTCHAN, HOWMANY)

INOUT reads HOWMANY words from channel INCHAN and writes them out on channel OUTCHAN. Each channel must be open in a mode between 8 and 12. On return, the EOF variables for the two channels will be the same as if ARRYIN & ARRYOUT had been used. If HOWMANY is less than zero, then transfer of data will cease only upon end of file or a device error. INOUT is not available in TENEX Sail.

## GETSTS, SETSTS

SETSTS (CHAN, NEW_STATUS);

issues a SETSTS uuo on channel CHAN with the status value NEW_STATUS.

STATUS ← GETSTS (CHAN)

returns the results of a GETSTS uuo on channel CHAN.

These functions do not exist in TENEX Sail. Instead, see GTSTS, GDSTS, STSTS, and SDSTS for analogous features.

## MTAPE

MTAPE (CHANNEL, MODE)

MTAPE is ignored unless the device associated with CHANNEL is a magnetic tape drive. It performs tape actions as follows:

| MODE | FUNCTION |
|---|---|
| "A" | Advance past one tape mark (or file) |
| "B" | Backspace past one tape mark |
| "E" | Write tape mark |
| "F" | Advance one record |
| "I" | Set 'IBM compatible' mode |
| "R" | Backspace one record |
| "S" | Write 3 inches of blank tape |
| "T" | Advance to logical end of tape |
| "U" | Rewind and unload |
| "W" | Rewind tape |
| NULL | Wait until all activity ceases |

## USETI, USETO

```
USETI (CHANNEL, VALUE);
USETO (CHANNEL, VALUE)
```

These routines are for random file access (see [SysCall]).

## REALIN, INTIN

```
VALUE ← REALIN (CHANNEL);
VALUE ← INTIN (CHANNEL)
```

Number input may be obtained using the functions REALIN or INTIN, depending on whether a Real number or an Integer is required. Both functions use the same free field scanner, and take as argument a channel number.

Free field scanning works as follows: characters are scanned one at a time from the input channel, ignoring everything until a digit or decimal point is encountered. Then a number is scanned according to this syntax, with zero bytes, line numbers, and carriage returns (but not linefeeds) ignored:

```
<number>
     ::= <sign> <real number>

<real number>
     ::= <decimal number>
     ::= <decimal number>  <exponent>
     ::= <exponent>

<decimal number>
     ::= <integer>
     ::= <integer> .
     ::= <integer> . <integer>
     ::= . <integer>

<integer>
     ::= <digit>
     ::= <integer> <digit>

<exponent>
     ::= @ <sign> <integer>
     ::= E <sign> <integer>

<sign>
     ::= +
     ::= -
     ::= <empty>
```

If the digit is not part of a number an error message will be printed and the program will halt. Typing a carriage return will cause the input function to return zero.

On input, leading zeros are ignored. The ten most significant digits are used to form the number. A check for overflow and underflow is made and an error message printed if this occurs. When using INTIN any exponent is removed by scaling the Integer number. Rounding is used in this process. All numbers are accurate to one half of the least significant bit.

After scanning the number the last delimiter is replaced on the input string and is returned as the break character for the channel. If no number is found, a zero is returned, and the break variable is set to -1; If an end of file or enabled error is sensed this is also returned in the appropriate channel variable. The maximum character count appearing in the OPEN call is ignored.

## REALSCAN, INTSCAN

```
VALUE ← REALSCAN (@"NUMBER_STRING",
                  @BRCHAR);
VALUE ← INTSCAN (@"NUMBER_STRING",
                 @BRCHAR)
```

These functions are identical in function to REALIN and INTIN. Their inputs, however, are obtained from their NUMBER_STRING arguments. These routines replace NUMBER_STRING by a string containing all characters left over after the number has been removed from the front.

## TMPIN, TMPOUT

```
"RESULT" ← TMPIN ("FILE", @ERRFLAG);
TMPOUT ("FILE", "TEXT", @ERRFLAG)
```

These routines do input and output to tmpcor files (simulated files kept in core storage--see [SysCall]).

TMPIN returns a string consisting of the entire contents of the tmpcor file of the specified name. Only the first three characters in the file name are significant. If the input fails for some reason (most likely: no tmpcor file with the specified name) then ERRFLAG is set to true and NULL is returned. Otherwise ERRFLAG is set to false.

TMPOUT writes its string argument into the specified tmpcor file. The ERRFLAG has the same function as in TMPIN; in case of error, the tmpcor file is not written. Likely causes for error are running out of tmpcor space (currently, the sum of the sizes of all the tmpcor files for a single job may not exceed =256 words) or attempting to write a null tmpcor file (i.e., calling TMPOUT with the string argument NULL).

TMPIN executes a TMPCOR uuo with code 1, and hence does not delete the specified tmpcor file. The length of the returned string will always be a multiple of five, since words rather than characters are actually being transferred. TMPOUT executes a TMPCOR uuo with code 3. The last word of the string is padded with nulls if necessary before the data transfer is done.

Neither function is available in TENEX Sail.

### AUXCLR, AUXCLV

RSLT ← AUXCLR (PORT, @ARG, FUNCTION);
RSLT ← AUXCLV (PORT, ARG, FUNCTION)

(TYMSHARE only.) These functions perform AUXCAL system calls; the only difference is whether ARG is by reference or by value. _SKIP_ is set.

### CHNIOR, CHNIOV

RSLT ← CHNIOR (CHAN, @ARG, FUNCTION);
RSLT ← CHNIOV (CHAN, ARG, FUNCTION)

(TYMSHARE only.) These functions perform CHANIO system calls; the only difference is whether ARG is by reference or by value. _SKIP_ is set.

## 6.5 TTY and PTY Routines

### TELETYPE I/O ROUTINES

Each of the I/O functions uses the TTCALL UUO's to do direct TTY I/O.

**BACKUP**
: The system attempts to back up its TTY input buffer pointer to the beginning of the last "line", thus allowing you to reread it. In general this cannot possibly work, so do not use BACKUP.

**CLRBUF**
: Flushes the input buffer.

**CHAR ← INCHRS**
: Returns a negative value if no characters have been typed; otherwise it is INCHRW.

**CHAR ← INCHRW**
: Waits for a character to be typed and returns that character.

**"STR" ← INCHSL (@FLAG)**
: Returns NULL with FLAG ≠ 0 if no lines have been typed. Otherwise it sets FLAG to 0 and performs INCHWL.

**"STR" ← INCHWL**
: Waits for a line to be typed and returns a string containing all characters up to (but not including) the activation character. The activation character is put into _SKIP_. If the activation character is CR then the next character is discarded (on the assumption that it is LF).

**"STR" ← INSTR (BRCHAR)**
: Returns as a string all characters up to, but not including, the first instance of BRCHAR. The BRCHAR instance is lost.

**"STR" ← INSTRL (BRCHAR)**
: Waits for a line to be typed, then performs INSTR.

"STR" ← INSTRS (@FLAG, BRCHAR)
Is INCHSL if no lines are waiting; INSTRL otherwise.

IONEOU (CHAR)
(TYMSHARE only.) The low-order 8 bits of CHAR are sent to the TTY in image mode.

OUTCHR (CHAR)
Types its character argument (right-justified in an integer variable).

OUTSTR ("STR")
Types its string argument until the end of the string or a null character is reached.

"STR" ← TTYIN (TABLE, @BRCHAR)
Uses the break table features described in page 36 and page 39 to return a string and break character. Mode "R" is illegal; line number modes are irrelevant. The input count (see page 33) is set at 100.

"STR" ← TTYINL (TABLE, @BRCHAR)
Waits for a line to be typed, then does TTYIN.

"STR" ← TTYINS (TABLE, @BRCHAR)
Sets BRCHAR to ≠0 and returns NULL if no lines are waiting. Otherwise it is TTYINL.

OLDVAL ← TTYUP (NEWVAL)
Causes conversion of lower case characters (a-z) to their upper case equivalents for strings read by any of the Sail teletype routines that do not use break tables. If NEWVAL is TRUE then conversion will take place on all subsequent inputs until TTYUP(FALSE) is called. OLDVAL will be set to the previous value of the conversion flag. If TTYUP has never been called, then no conversions will take place, and the first call to TTYUP will return FALSE. In TENEX, TTYUP sets the system parameter using the STPAR jsys to convert to upper case.

## —— PSEUDO-TELETYPE FUNCTIONS ——

Pseudo-teletype functions are available at SUAI only.

LODED ("STR")
Loads the line editor with the string argument. PTOSTR should be used rather than LODED if possible, since LODED works only on a DD or III, while PTOSTR works on all terminals.

"STR" ← PTYALL (LINE)
Returns whatever is in the PTY's output buffer. No waiting is done.

CHAR ← PTCHRS (LINE)
Reads a character from the PTY if there is one, returns -1 if none.

CHAR ← PTCHRW (LINE)
Waits for a character from the PTY and returns it.

PTOCHS (LINE, CHAR)
Tries to send a character to a PTY. If the attempt was successful, the global variable _SKIP_ is -1, otherwise 0.

PTOCHW (LINE, CHAR)
Sends a character to a PTY, waiting if necessary.

NUMBER ← PTOCNT (LINE)
Returns the number of free characters in the PTY output buffer.

NUMBER ← PTIFRE (LINE)
Returns the number of free characters in the PTY input buffer.

PTOSTR (LINE, "STR")
Sends the string to the PTY, waiting if necessary. PTOSTR (0, "STR") sends the string to your TTY.

LINE ← PTYGET
Gets a new pseudo-teletype line number and returns it. The global variable _SKIP_ is -1 if the attempt to get a PTY was successful, and 0 otherwise.

**CHARACTERISTICS ← PTYGTL (LINE)**
Returns line characteristics for the PTY.

**"STR" ← PTYIN (LINE, BKTBL, @BRCHAR)**
Reads from the PTY (waiting if necessary) according to break table conventions. The break character is stored in BRCHAR.

**PTYREL (LINE)**
Releases PTY identified by LINE.

**PTYSTL (LINE, CHARACTERISTICS)**
Sets line characteristics for the PTY specified by LINE.

**"STR" ← PTYSTR (LINE, BRCHAR)**
Reads characters from the PTY, waiting if necessary, until a character equal to BRCHAR is seen. All but the break character is returned as the string. If the break character was '15 (carriage return), the following character is snarfed (on the assumption that it is a linefeed).

## 6.6 Example of TOPS-10 I/O

```
BEGIN "COPY"
COMMENT copies a text file, inserting a semicolon at the
beginning of each line, deleting SOS line numbers and
zero bytes, if any.  Prints the page number as it goes;

REQUIRE "[][]" DELIMITERS;
DEFINE CRLF=[('15&'12)],LF=['12], FF=['14];
INTEGER COLONTAB;

RECORD_CLASS $FILE (STRING DEVICE, NAME;
  INTEGER CHANNEL, MODE, IBUF, OBUF,
    COUNT, BRCHAR, EOF, LINNUM, PAGNUM, SOSNUM);

RECORD_POINTER($FILE) PROCEDURE OPENUP
  (STRING FILNAM; INTEGER MODE, IBUF, OBUF);
BEGIN "OPENUP"
STRING T; RECORD_POINTER ($FILE) Q; INTEGER BRK;
Q←NEW_RECORD ($FILE); T←SCAN (FILNAM, COLONTAB, BRK);
$FILE:DEVICE[Q]←(IF BRK=":" THEN T ELSE "DSK");
$FILE:NAME[Q]←(IF BRK=":" THEN FILNAM ELSE T);
$FILE:MODE[Q]←MODE; $FILE:IBUF[Q]←IBUF;
$FILE:OBUF[Q]←OBUF; OPEN ($FILE:CHANNEL[Q]←GETCHAN,
  $FILE:DEVICE[Q], MODE, IBUF, OBUF, $FILE:COUNT[Q],
  $FILE:BRCHAR[Q], $FILE:EOF[Q]←-1);
IF NOT($FILE:EOF[Q]) THEN BEGIN
  SETPL ($FILE:CHANNEL[Q], $FILE:LINNUM[Q],
    $FILE:PAGNUM[Q], $FILE:SOSNUM[Q]);
  IF IBUF THEN
    LOOKUP ($FILE:CHANNEL[Q], $FILE:NAME[Q], $FILE:EOF[Q]);
  IF OBUF AND NOT ($FILE:EOF[Q]) THEN
    ENTER ($FILE:CHANNEL[Q], $FILE:NAME[Q], $FILE:EOF[Q]);
END;
$FILE:PAGNUM[Q]←1;
IF $FILE:EOF[Q] THEN RELEASE($FILE:CHANNEL[Q]);
RETURN(Q)
END "OPENUP";
COMMENT Sail I/O should be rewritten to do this ↑;

RECORD_POINTER ($FILE) PROCEDURE GETFILE
  (STRING PROMPT; INTEGER MODE, I, O);
BEGIN "GETFILE"
RECORD_POINTER ($FILE) F; INTEGER REASON;
WHILE TRUE DO BEGIN "try"
  PRINT (PROMPT);
  IF (REASON←$FILE:EOF[F←OPENUP (INCHWL,
    MODE, I, O)])=0 THEN RETURN (F);
  IF REASON=-1 THEN
    PRINT ("Device ", $FILE:DEVICE[F], " not available.")
  ELSE PRINT("Error, ", CASE (0 MAX REASON MIN 4) OF
    ("no such file ", "illegal PPN ", "protection ",
      "busy ", "??? "), $FILE:NAME[F], CRLF);
END "try";
END "GETFILE";


RECORD_POINTER ($FILE) SRC, SNK;
INTEGER FFLFTAB;

SETBREAK (COLONTAB←GETBREAK, ":", "", "ISN");
WHILE TRUE DO BEGIN "big loop"
   STRING LINE;
   SRC←GETFILE ("Copy from:", 0, 5, 0);
   $FILE:COUNT[SRC]←200;
   SNK←GETFILE (" to:", 0, 0, 5);
   SETBREAK (FFLFTAB←GETBREAK, FF&LF, "", "INA");

   WHILE TRUE DO BEGIN "a line"
      LINE←INPUT ($FILE:CHANNEL[SRC], FFLFTAB);
      IF $FILE:EOF[SRC] THEN DONE;
      IF $FILE:BRCHAR[SRC]=FF THEN BEGIN
        PRINT (" ", $FILE:PAGNUM[SRC]);
        LINE←LINE&
          INPUT ($FILE:CHANNEL[SRC], FFLFTAB) END;
      CPRINT ($FILE:CHANNEL[SNK], ";", LINE)
   END "a line";
   RELEASE ($FILE:CHANNEL[SRC]);
   RELEASE ($FILE:CHANNEL[SNK])
END "big loop";
END "COPY"
```

# SECTION 7

# EXECUTION TIME ROUTINES

Please read Execution Time Routines in General, page 33, if you are unfamiliar with the format used to describe runtime routines.

## 7.1 Type Conversion Routines

### SETFORMAT

SETFORMAT (WIDTH, DIGITS)

This function allows specification of a minimum width for strings created by the functions CVS, CVOS, CVE, CVF, and CVG (see page 46 and following). If WIDTH is positive then enough blanks will be inserted in front of the resultant string to make the result at least WIDTH characters long. The sign, if any, will appear after the blanks. If WIDTH is negative then leading zeroes will be used in place of blanks. The sign, of course, will appear before the zeroes. The parameter WIDTH is initialized by the system to zero.

In addition, the DIGITS parameter allows one to specify the number of digits to appear following the decimal point in strings created by CVE, CVF, and CVG. This number is initially 7. See the writeups on these functions for details.

NOTE: All type conversion routines, including those that SETFORMAT applies to, are performed at compile time if their arguments are constants. However, Setformat does not have its effect until execution time. Therefore, CVS, CVOS, CVE, CVF, and CVG of constants will have no leading zeros and 7 digits (if any) following the decimal point.

### GETFORMAT

GETFORMAT (@WIDTH, @DIGITS)

The WIDTH and DIGIT settings specified in the last SETFORMAT call are returned in the appropriate reference parameters.

### CVS

"ASCII_STRING" ← CVS (VALUE);

The decimal Integer representation of VALUE is produced as an ASCII String with leading zeroes omitted (unless WIDTH has been set by SETFORMAT to some negative value). "-" will be concatenated to the String representing the decimal absolute value of VALUE if VALUE is negative.

### CVD

VALUE ← CVD ("ASCII_STRING")

ASCII_STRING should be a String of decimal ASCII characters perhaps preceded by plus and/or minus signs. Characters with ASCII values ≤ SPACE ('40) are ignored preceding the number. Any character not a digit will terminate the conversion (with no error indication). The result is a (signed) integer.

### CVOS

"ASCII_STRING" ← CVOS (VALUE)

The octal Integer representation of VALUE is produced as an ASCII String with leading zeroes omitted (unless WIDTH has been set to some negative value by SETFORMAT. No "-" will be used to indicate negative numbers. For instance, -5 will be represented as "777777777773".

### CVO

VALUE ← CVO ("ASCII_STRING")

This function is the same as CVD except that the input characters are deemed to represent Octal values.

### CVE, CVF, CVG

"STRING" ← CVE (VALUE);
"STRING" ← CVF (VALUE);
"STRING" ← CVG (VALUE)

Real number output is facilitated by means of one of three functions CVE, CVG, or CVF, corresponding to the E, G, and F formats of *FORTRAN IV. Each of these functions* takes as argument a real number and returns a string. The format of the string is controlled by another function SETFORMAT (WIDTH, DIGITS) (see page 46) which is used to change WIDTH from zero and DIGITS from 7, their initial values. WIDTH specifies the minimum string length. If WIDTH is positive leading blanks will be inserted and if negative leading zeros will be inserted.

The following table indicates the strings returned for some typical numbers. _ indicates a space and it is assumed that WIDTH←10 and DIGITS←3.

| CVF | CVE | CVG |
|---|---|---|
| _____.000 | __.100@-3_ | __.100@-3_ |
| _____.001 | __.100@-2_ | __.100@-2_ |
| _____.010 | __.100@-1_ | __.100@-1_ |
| _____.100 | __.100____ | __.100____ |
| _____1.000 | __.100@1__ | __1.00____ |
| ____10.000 | __.100@2__ | __10.0____ |
| ___100.000 | __.100@3__ | __100.____ |
| __1000.000 | __.100@4__ | __.100@4__ |
| _10000.000 | __.100@5__ | __.100@5__ |
| _100000.000 | __.100@6__ | __.100@6__ |
| _1000000.000 | __.100@7__ | __.100@7__ |
| -1000000.000 | _-.100@7__ | _-.100@7__ |

The first character ahead of the number is either a blank or a minus sign. With WIDTH←-10 plus and minus 1 would print as:

| CVF | CVE | CVG |
|---|---|---|
| _00001.000 | _0.100@1__ | _01.00____ |
| -00001.000 | -0.100@1__ | -01.00____ |

All numbers are accurate to one unit in the eighth digit. If DIGITS is greater than 8, trailing zeros are included; if less than eight, the number is rounded.

### CVASC, CVASTR, CVSTR

VALUE ← CVASC ("STRING");
"STRING" ← CVASTR (VALUE);
"STRING" ← CVSTR (VALUE)

These routines convert between a Sail String and an integer containing 5 ASCII characters left justified in a 36-bit word; the extra bit is made zero (CVASC) or ignored (CVASTR, CVSTR). CVASC converts from String to ASCII. Both CVSTR and CVASTR convert from a word of ASCII to a string. CVSTR always returns a string of length five, while CVASTR stops converting at the first null ('0) character.

CVASTR (CVASC ("ABC")) is "ABC"
CVSTR (CVASC ("ABC")) is "ABC" & 0 & 0

### CV6STR, CVSIX, CVXSTR

"STRING" ← CV6STR (VALUE);
VALUE ← CVSIX ("STRING");
"STRING" ← CVXSTR (VALUE)

The routines CV6STR, CVSIX, and CVXSTR are the SIXBIT analogues of CVASTR, CVASC, and CVSTR, respectively. The character codes are converted, ASCII in the String ↔ SIXBIT in the integer. CVXSTR always returns a string of length six, while CV6STR stops converting upon reaching a null character.

CV6STR (CVSIX ("XYZ")) is "XYZ", not "XYZ ".
CV6STR (CVSIX ("X Y Z")) is "X", not "X Y Z" or "XYZ".

## 7.2 String Manipulation Routines

### EQU

VALUE ← EQU ("STR1", "STR2")

The value of this function is TRUE if STR1 and STR2 are equal in length and have identically the same characters in them (in the same order). The value of EQU is FALSE otherwise.

### LENGTH

VALUE ← LENGTH ("STRING")

LENGTH is always an integer-valued function. If the argument is a String, its length is the number of characters in the string. The length of an algebraic expression is always 1 (see page 23). LENGTH is usually compiled in line.

### LOP

VALUE ← LOP (@STRINGVAR)

The LOP operator applied to a String variable removes the first character from the String and returns it in the form given in page 23 above. The String no longer contains this character. LOP applied to a null String has a zero value. LOP is usually compiled in line. LOP may not appear as a statement.

### SUBSR, SUBST

"RSLT" ← *SUBSR* ("*STRING*", *LEN*, *FIRST*);
"RSLT" ← SUBST ("STRING", LAST, FIRST)

These routines are the ones used for performing substring operations. SUBSR (STR, LEN, FIRST) is STR[FIRST FOR LEN] and SUBST (STR, LAST, FIRST) is STR[FIRST TO LAST].

## 7.3 Liberation-from-Sail Routines

### CODE

RESULT ← CODE (INSTR, @ADDR)

This function is equivalent to the FAIL statements:

```
EXTERNAL    .SKIP.  ;DECLARE AS _SKIP_ IN SAIL
SETOM       .SKIP.  ;ASSUME SKIP
MOVE        0,INSTR
ADDI        0,@ADDR
XCT         0
SETZM       .SKIP.  ;DIDN'T SKIP
RETURN      (1)
```

In other words, it executes the instruction formed by adding the address of the ADDR variable (passed by reference) to the number INSTR. Before the operation is carried out, AC1 is loaded from a special cell (initially 0). AC1 is returned as the result, and also stored back into the special cell after the instruction is executed. The global variable _SKIP_ (.SKIP. in DDT or FAIL) is FALSE (0) after the call if the executed instruction did not skip; TRUE (currently -1) if it did. Declare this variable as EXTERNAL INTEGER _SKIP_ if you want to use it.

### CALL

RESULT ← CALL (VALUE, "FUNCTION")

This function is equivalent to the FAIL statements:

```
EXTERNAL    .SKIP.
SETOM       .SKIP.
MOVE        1,VALUE
CALL        1,(SIXBIT /FUNCTION/)
SETZM       .SKIP.          ;DID NOT SKIP
RETURN      (REGISTER 1)
```

TENEX users should see more on CALL, page 80.

### CALLI

RESULT ← CALLI (VALUE, FUNCTION)

(TYMSHARE only.) Like CALL, only CALLI.

### USERCON

USERCON (@INDEX, @VALUE, FLAG)

This function allows inspection and alteration of the "User Table". The user table is always loaded with your program and contains many interesting variables. Declare an index you are interested in as an External Integer (e.g., EXTERNAL INTEGER REMCHR). This will, when loaded, give an address which is secretly a small integer index into the User Table. When passed by reference, this index is available to

USERCON. The names and meanings of the various User Table indices can be found in the file HEAD, wherever Sail compiler program text files are sold.

USERCON always returns the current value of the appropriate User Table entry (the Global Upper Segment Table is used if FLAG is negative and your system knows about such things). If FLAG is odd, the contents of VALUE before the call replaces the old value in the selected entry of the selected table.

By now the incredible danger of this feature must be apparent to you. Be sure you understand the ramifications of any changes you make to any User Table value.

GOGTAB

Direct access to the user table can be gained by declaring EXTERNAL INTEGER ARRAY GOGTAB[0:n]; The clumsy USERCON linkage is obsolete.

The symbolic names of all GOGTAB entries can be obtained by requiring SYS:GOGTAB.DEF (<SAIL>GOGTAB.DEF on TENEX) as a source file. This file contains DEFINEs for all of the user table entries.

## USERERR

USERERR (VALUE, CODE, "MSG",
"RESPONSE"(NULL))

USERERR generates an error message. See page 138 for a description of the error message format. MSG is the error message that is printed on the teletype or sent to the log file. If CODE = 2, VALUE is printed in decimal on the same line. Then on the next line the "Last SAIL call" message may be typed which indicates where in the user program the error occurred. If CODE is 1 or 2, a "→" will be typed and execution will be allowed to continue. If it is 0, a "?" is typed, and no continuation will be permitted. The string RESPONSE, if included in the USERERR call, will be scanned before the input buffer is scanned. In fact, if the string RESPONSE satisfies the error handler, the input buffer will not be scanned at all. Examples:

USERERR (0, 1, "LINE TOO LONG"); Gives error message and allows continuation.

USERERR (0, 1, NULL, "QLA"); Resets mode of error handler to Quiet, Logging, and Automatic continuation. Then continues.

## ERMSBF

ERMSBF (NEWSIZE)

This routine insures that error messages of NEWSIZE characters can be handled. The error message buffer is initially 256 characters, which is sufficient for any Sail-generated error. USERERR can generate longer messages, however.

## EDFILE

EDFILE ("FILENAME", LINE, PAGE, BITS(0))

(Not on TENEX.) Exits to an editor. Which editor is determined by the bits which are on in the second parameter, LINE. If bit 0 or bit 1 (600000,,0 bits) is on, then LINE is assumed to be ASCID and SOS is called. If neither of these bits is on, then LINE is assumed to be of the form attach count,,sequential line number and E is called. PAGE is the binary page number. BITS defaults to zero and controls the editing mode.

0 edit
1 no directory (as in /N)
2 readonly (as in /R)
4 create (as in /C)

In addition, the accumulators are set up from INIACS (see below) so that the E command ∝X RUN will run the dump file from which the current program was gotten. [Accumulators 0 (file name), 1 (extension), and 6 (device) are loaded from the corresponding values in INIACS.]

## INIACS

The contents of locations 0-'17 are saved in

block INIACS when the core image is started for the first time. Declare INIACS as an external integer and use START_CODE or MEMORY[LOCATION(INIACS)+n] to reference this block.

## 7.4 Byte Manipulation Routines

### LDB, DPB, etc.

```
VALUE ← LDB (BYTE_POINTER);
VALUE ← ILDB (@ BYTE_POINTER);
DPB (BYTE, BYTE_POINTER);
IDPB (BYTE, @ BYTE_POINTER);
IBP  (@ BYTE_POINTER)
```

LDB, ILDB, DPB, IDPB, and IBP are Sail constructs used to invoke the PDP-10 byte loading instructions. The arguments to these functions are expressions which are interpreted as byte pointers and bytes. In the case of ILDB, IDPB, and IBP, you are required to use an algebraic variable as argument as the *byte_pointer*, so that the byte pointer (i.e. that algebraic variable) may be incremented.

### POINT

```
VALUE ← POINT (BYTE SIZE,
      @EFFECTIVE ADDRESS, LAST BIT NUMBER)
```

POINT returns a byte pointer (hence it is of type integer). The three arguments correspond exactly to the three arguments to the POINT pseudo-op in FAIL.

## 7.5 Other Useful Routines

### CVFIL

```
VALUE ← CVFIL ("FILE_SPEC", @EXTEN, @PPN)
```

FILE_SPEC has the same form as a file name specification for LOOKUP or ENTER. The SIXBIT for the file name is returned in VALUE. SIXBIT values for the extension and project-programmer numbers are returned in the respective reference parameters. Any unspecified portions of the FILE_SPEC will result in zero values. The global variable _SKIP_ will be 0 if no errors occurred, non-zero if an invalid file name specification is presented.

### FILEINFO

```
FILEINFO (@INFOARRAY)
```

FILEINFO fills the 6-word array INFOARRAY with the following six words from the most recent LOOKUP, ENTER, or RENAME:

```
FILENAME
EXT„(2)hidate2 (15)date1
(9)prot (4)Mode (11)time (12)lodate2
negative swapped word count
0 (unless opened in magic mode)
0
```

See [SysCall]; TENEX users should use JFNS instead.

### ARRINFO

```
VALUE ← ARRINFO (ARRAY, PARAMETER)
```

| | |
|---|---|
| ARRINFO (ARRAY, -1) | is the number of dimensions for the array. This number is negative for String arrays. |
| ARRINFO (ARRAY, 0) | is the total size of the array in words. |
| ARRINFO (ARRAY, 1) | is the lower bound for the first dimension. |
| ARRINFO (ARRAY, 2) | is the upper bound for the first dimension. |
| ARRINFO (ARRAY, 3) | is the lower bound for the second dimension. |
| ARRINFO (... | etc. |

### ARRBLT

ARRBLT (@DEST, @SOURCE, NUM)

NUM words are transferred (using BLT) from consecutive locations starting at SOURCE to consecutive locations starting at DEST. No bounds checking is performed. This function does not work well for String Arrays (nor set nor list arrays).

### ARRTRAN

ARRTRAN (DESTARR, SOURCEARR)

This function copies information from SOURCEARR to DESTARR. The transfer starts at the first data word of each array. The minimum of the sizes of SOURCEARR and DESTARR is the number of words transferred.

### ARRCLR

ARRCLR (ARRAY, VALUE(0))

This routine stores VALUE into each element of ARRAY. The most common use is with VALUE omitted, which clears the array; i.e., arithmetic arrays get filled with zeros, string arrays with NULLs, itemvar arrays with ANYs, record_pointer arrays with NULL_RECORD. One may use ARRCLR with set and list arrays, but the set and list space will be lost (i.e., un-garbage-collectible). Do not supply anything other than 0 (0, NULL, PHI, NIL, NULL_RECORD) for VALUE when clearing a string, set, list, or record_pointer array unless you know what you are doing. Using a real value for an itemvar array is apt to cause strange results. (If you use an integer then ARRAY will be filled with CVI (value).)

### IN_CONTEXT

VALUE ← IN_CONTEXT (VARI, CONTXT)

IN_CONTEXT is a boolean which tells one if the specified variable is in the specified context. VARI may be any variable, array element, array name, or Leap variable. If that variable, element or array was REMEMBERed in that context, IN_CONTEXT will return True. IN_CONTEXT will also return true if VARI is an array element and the whole array was Remembered in that context (by using REMEMBER <array_name>). On the other hand, if VARI is an array name, then IN_CONTEXT will return true only if one has Remembered that array with a REMEMBER <array_name>.

### CHNCDB

VALUE ← CHNCDB (CHANNEL)

(Not on TENEX.) This integer procedure returns the address of the block of storage which Sail uses to keep track of the specified channel. It is provided for the benefit of assembly language procedures that may want to do I/O inside some fast inner loop, but which may want to live in a Sail core image & use the Sail OPEN, etc.

## 7.6 Numerical Routines

These numerical routines are new as predeclared runtimes in Sail. The routines themselves are quite standard.

The standard trigonometric functions. ASIN, ACOS, ATAN and ATAN2 return results in radians. The ATAN2 call takes arc-tangent of the quotient of its arguments; in this way, it correctly preserves sign information.

```
REAL PROCEDURE SIN (REAL RADIANS);
REAL PROCEDURE COS (REAL RADIANS);
REAL PROCEDURE SIND (REAL DEGREES);
REAL PROCEDURE COSD (REAL DEGREES);

REAL PROCEDURE ASIN (REAL ARGUMENT);
REAL PROCEDURE ACOS (REAL ARGUMENT);
REAL PROCEDURE ATAN (REAL ARGUMENT);
REAL PROCEDURE ATAN2 (REAL NUM, DEN)
```

The hyperbolic trigonometric functions.

```
REAL PROCEDURE SINH (REAL ARGUMENT);
REAL PROCEDURE COSH (REAL ARGUMENT);
REAL PROCEDURE TANH (REAL ARGUMENT)
```

The square-root function:

```
REAL PROCEDURE SQRT (REAL ARGUMENT)
```

A pseudo-random number generator. The argument specifies a new value for the seed (if the argument is 0, the old seed value is used. Thus to get differing random numbers, this argument should be zero.) Results are normalized to lie in the range [0,1].

```
REAL PROCEDURE RAN (INTEGER SEED)
```

Logarithm and exponentiation functions. These functions are the same ones used by the Sail exponentiation operator. The base is e (2.71828182845904). The logarithm to the base 10 of e is 0.4342944819.

```
REAL PROCEDURE LOG (REAL ARGUMENT);
REAL PROCEDURE EXP (REAL ARGUMENT)
```

These functions may occasionally be asked to compute numbers that lie outside the range of legal floating-point numbers on the PDP-10. In these cases, the routines issue sprightly error messages that are continuable.

OVERFLOW
In order to better perform their tasks, these routines enable the system interrupt facility for floating-point overflow and underflow errors. If an underflow is detected, the results are set to 0 (a feat not done by the PDP-10 hardware, alas). Be aware that such underflow fixups will be done to every underflow that occurs in your program. For further implementation details, see the section below.

If you would like to be informed of any numerical exceptions, you can call the runtime:

```
TRIGINI (LOCATION (simple-procedure-name))
```

Every floating-point exception that is not expected by the interrupt handler (the numerical routines use a special convention to indicate that arithmetic exception was expected) will cause the specified simple procedure to be called. This procedure may look around the world as described for 'export' interrupt handlers, page 120. If no TRIGINI call is done, the interrupt routine will simply dismiss unexpected floating-point interrupts.

ENTRY POINTS
In order to avoid confusion (by the loader) with older trig packages, the entry points of the Sail arithmetic routines all have a "S" appended to the end. Thus, SIN has the entry point SINS, etc. WARNING: If a program plans to use the Sail intrinsic numerical routines, it should NOT include external declarations to them, since this will probably cause the FORTRAN library routines to be loaded.

OVERFLOW IMPLEMENTATION
This section may be skipped by all but those interested in interfacing number crunching assembly code (where overflow and underflow are expected to be a problem) with Sail routines.

The Sail arithmetic interrupt routines first check to see if the interrupt was caused by floating exponent underflow. If it was, then the result is set to zero, be it in an accumulator, memory, or both. Then if the arithmetic instruction that caused the interrupt is followed by a JFCL, the AC field of the JFCL is compared with the PC flag bits to see if the JFCL tests for any of the flags that are on. If it does, those flags are cleared and the program proceeds at the effective address of the JFCL (i.e., the hardware is simulated in that case). Note that no instructions may intervene between the interrupt-causing instruction and the JFCL or the interrupt routines will not see the JFCL. They only look one instruction ahead. Note that in any case, floating exponent underflow always causes the result to be set to zero. There is no way to disable that effect.

# SECTION 8

# PRINT

## 8.1 Syntax

<print_statement>
    ::= PRINT ( <expression_list> )
    ::= CPRINT ( <integer_expression> ,
        <expression_list> )

## 8.2 Semantics

The new constructs PRINT and CPRINT are conveniences for handling character output. Code which formerly looked like

```
OUTSTR ("The values are " & CVS (I) & " and " &
    CVG (X) & " for item " & CVIS (IT, JUNK));
```

may now be written

```
PRINT ("The values are ", I, X, " for item ", IT);
```

The first expression in <expression_list> is evaluated, formatted as a string, and routed to the appropriate destination. Then the second expression is evaluated, formatted, and dispatched; etc. (If an expression is an assignment expression or a procedure call then side effects may occur.)

**DEFAULT FORMATS**
String expressions are simply sent to the output routine. Integer expressions are first sent to CVS, and Real expressions are passed to CVG; the current SETFORMAT parameters are used. Item expressions use the print name for the item if one exists, otherwise ITEM!nnnn, where nnnn is the item number. Sets and lists show their item components separated by commas. Sets are surrounded by single braces and lists by double braces. PHI and NIL are printed for the empty set and empty list respectively. Record pointers are formatted as the name of the record class, followed by a ".", followed by the (decimal) address of the record. NULL!RECORD is printed for the empty record.

If the default format is not satisfactory then the user may give a function call as an argument. For example,

```
PRINT (CVOS (I));
```

will print I in octal, since CVOS is called first. (The expression CVOS (I) is of course a String expression.) Wizards may also change the default formatting function for a given syntactic type.

**DESTINATIONS**
CPRINT interprets <integer_expression> as a Sail channel number and sends all output to that channel. The following two statements are functionally equivalent:

```
CPRINT (CHAN, "The values are ", I, " and ", X);
OUT (CHAN, "The values are "&CVS (I)&" and "&CVG (X));
```

PRINT initially sends all output to the terminal but can also direct output to a file or any combination of terminal and/or file. The modes of PRINT are (dynamically) established and queried by SETPRINT and GETPRINT.

——————— SETPRINT, GETPRINT ———————

```
SETPRINT ("FILE_NAME", "MODE");
"MODE" ← GETPRINT
```

Here MODE is a single character which represents the destination of PRINT output.

| MODE | MEANING |
|---|---|
| "T" | the Terminal gets all PRINT output. If an output file is open then close it. "T" is the mode in which PRINT is initialized. |
| "F" | File gets PRINT output. If no file is open then open one as described below. |
| "B" | Both terminal and file get PRINT output. If no file is open then open one as described below. |
| "N" | Neither the file nor the terminal gets any output. If a file is open then close it. |
| "S" | Suppress all output, but open a file if none is open. |

"O" a file is Open, but the terminal is getting all output. If no file is open then open one as described below.

"C" the terminal gets output, but ignore whether or not a file is open and whether or not it is getting output.

"I" terminal does not get output. Ignore whether or not a file is open and whether or not file is getting any output.

The first 6 possibilities represent the logical states of the PRINT system and are the characters which GETPRINT can return. The "C" and "I" modes turn terminal output on and off without disturbing anything else. The PRINT statement is initialized to mode "T" -- print to Terminal. Modes "T", "F", and "B" are probably the most useful. The other modes are included for completeness and allow the user to switch between various combinations dynamically.

If SETPRINT is called in such a way that a file has to be opened -- e.g., mode "F" and no file is open -- then FILE_NAME will be used as the name of the output file. If FILE_NAME is NULL then the filename will be obtained from the terminal.

```
SETPRINT (NULL, "F");
```

first types the message

```
File for PRINT output •
```

and uses the response as the name of a file to open. On TENEX, GTJFN with recognition is used; on TOPS-10 and its variants the filename is read with INCHWL. The file opened by SETPRINT will be closed when the program terminates by falling through the bottom. It will also be closed if the user calls SETPRINT with some mode that closes the file -- e.g., "T" will close an output file if one is open.

SETPRINT and GETPRINT are related only to PRINT; they have no effect on CPRINT.

## SIMPLE USE

Here are a few examples of common output situations.

1) PRINT to TERMINAL. Simply use PRINT; do not bother with SETPRINT.

2) PRINT to FILE. Call SETPRINT (NULL, "F"); and type the name of the output file when it asks.

3) PRINT to FILE and TERMINAL. At the beginning of the program call SETPRINT (NULL, "B"); and type the name when asked.

4) PRINT to FILE always and sometimes also to TERMINAL. Use SETPRINT (NULL, "B"); and give the name of the file when it asks. This sets output to both the terminal and the file. Then to ignore the terminal (leaving the file alone), call SETPRINT (NULL, "I"); To resume output at the terminal use SETPRINT (NULL, "C"); This is useful for obtaining a cleaned-up printout on the file with error messages going to the terminal.

## CAVEATS

Trying to exploit the normal Sail type conversions will probably lead to trouble with PRINT and CPRINT. Printing single ASCII characters is a particular problem.

```
OUTSTR ('14);
```

prints a form-feed onto the terminal , but

```
PRINT ('14);
```

prints "12". The reason, of course, is the default formatting of integers by PRINT or CPRINT. This problem is particularly severe with macros that have been defined with an integer to represent an ASCII character. For example,

```
DEFINE TAB="'11";
PRINT (TAB);
```

will print "9". The solution is to define the macro so that it expands to a STRING constant rather than an integer.

```
DEFINE TAB=c"      "ɔ; or
DEFINE TAB=c('11 & NULL)ɔ;
```

Also, remember that the first argument to CPRINT is the channel number.

FOR WIZARDS ONLY
All output going to either the PRINT or CPRINT statements can be trapped by setting user table entry $$PROU to the address of a SIMPLE procedure that has one string and one integer argument.

```
SIMPLE PROCEDURE MYPRINT
    (INTEGER CHAN; STRING S);
BEGIN ... END;

GOGTAB[$$PROU] ← LOCATION (MYPRINT);
```

The CHAN argument is either the CHAN argument for CPRINT, or -1 for PRINT. If this trap is set then all output from PRINT and CPRINT goes through the user routine and is not printed unless the user invokes OUT or OUTSTR from within the trap routine itself.

To trap the formatting function for any syntactic type the user should set the appropriate user table address to the location of a function that returns a string and takes as an argument the syntactic type in question. To print integers in octal , preceded by "", use

```
SIMPLE STRING PROCEDURE MYCVOS (INTEGER I);
RETURN ("" & CVOS (I));

GOGTAB[$$FINT] ← LOCATION (MYCVOS);
```

The names for the addresses in the user table associated with each formatting function are:

| INDEX | TYPE |
|---|---|
| $$FINT | INTEGER |
| $$FREL | REAL |
| $$FITM | ITEM |
| $$FSET | SET |
| $$FLST | LIST |
| $$FSTR | STRING |
| $$FREC | RECORD_POINTER |

To restore any formatting function to the default provided by the PRINT system, zero the appropriate entry of the user table.

# SECTION 9

# MACROS AND CONDITIONAL COMPILATION.

## 9.1 Syntax

```
<define>
     ::= DEFINE <def_list> ;
     ::= REDEFINE <def_list> ;
     ::= EVALDEFINE <def_list> ;
     ::= EVALREDEFINE <def_list> ;

<def_list>
     ::= <def>
     ::= <def_list> , <def>

<def>
     ::= <identifier> = <macro_body>
     ::= <identifier> ( <id_list> ) =
           <macro_body>
     ::= <identifier> <string_constant> =
           <macro_body>
     ::= <identifier> ( <id_list> )
           <string_constant> = <macro_body>

<macro_body>
     ::= <delimited_string>
     ::= <constant_expression>
     ::= <macro_body> & <macro_body>

<macro_call>
     ::= <macro_identifier>
     ::= <macro_identifier>
           ( <macro_param_list> )
     ::= <macro_identifier> <string_constant>
           ( <macro_param_list> )

<macro_identifier>
     ::= <identifier>

<macro_param_list>
     ::= <macro_param>
     ::= <macro_param_list> , <macro_param>

<cond_comp_statement>
     ::= <conditional_c.c.s.>
     ::= <while_c.c.s.>
     ::= <for_c.c.s.>
     ::= <for_list_c.c.s.>
     ::= <case_c.c.s.>

<conditional_c.c.s.>
     ::= IFC <constant_expression> THENC
           <anything> ENDC
     ::= IFC <constant_expression> THENC
           <anything> ELSEC <anything> ENDC
     ::= IFCR <constant_expression> THENC
           <anything> ENDC
     ::= IFCR <constant_expression> THENC
           <anything> ELSEC <anything> ENDC

<while_c.c.s.>
     ::= WHILEC <delimited_expr> DOC
           <delimited_anything> ENDC

<for_c.c.s.>
     ::= FORC <identifier> ←
           <constant_expression> STEPC
           <constant_expression> UNTILC
           <constant_expression> DOC
           <delimited_anything> ENDC

<for_list_c.c.s.>
     ::= FORLC <identifier> ←
           ( <macro_param_list> ) DOC
           <delimited_anything> ENDC

<case_c.c.s.>
     ::= CASEC <constant_expression> OFC
           <delimited_anything_list> ENDC

<delimited_anything_list>
     ::= <delimited_anything>
     ::= <delimited_anything_list> ,
           <delimited_anything>

<assignc>
     ::= ASSIGNC <identifier> = <macro_body> ;
```

<delimited_string>, <macro_param>, <delimited_expr>, <anything> and <delimited_anytning> are explained in the following text.

## 9.2 Delimiters

There are two types of delimiters used by the Sail macro scanner: macro body delimiters and macro parameter delimiters. Their usage will be precisely defined in the sections on Macro Bodies and Parameters to Macros. Here we will discuss their declaration and scope, which is very important when using source files with different delimiters (see page 11 to find out about source files).

Sail initializes both left and right delimiters of both body and parameter delimiters to the double quote ("). One may change delimiters by saying

```
REQUIRE "⊂⊃<>" DELIMITERS.
```

In this example, the left and right body delimiters become "⊂" and "⊃", while the left and right parameter delimiters become "<" and ">". Require Delimiters may appear wherever a statement or declaration is legal. One should Require Delimiters whenever all but the most simple macros are going to be used. The first Require Delimiters will initialize the macro facility; if this is not done, some of the following conveniences will not exist and only very simple macros like defining CRLF = "('12 & '15 )" may be done.

Delimiters do not follow block structure. They persist until changed. Furthermore, each time new delimiters are Required, they are stacked on a special "delimiters stack". The old delimiters may be revived by saying

```
REQUIRE UNSTACK_DELIMITERS
```

Thus, each source file with macros should begin with a Require delimiters, and end with an Unstack_delimiters. It is impossible to Unstack off the bottom of the stack. The bottom element of the stack is the double quote delimiters that Sail initialized the program to. If you Unstack from these, the Unstack will become a no-op, and the double quote delimiters remain the delimiters of your program.

One may circumvent the delimiter stacking feature by saying

```
REQUIRE "⊂⊃<>" REPLACE_DELIMITERS
```

instead of REQUIRE "⊂⊃<>" DELIMITERS. This doesn't deactivate the stacking feature, it merely changes the active delimiters without stacking them.

To revert to the primitive, initial delimiter mode where double quotes are the active delimiters, one may say

```
REQUIRE NULL DELIMITERS
```

Null delimiters are stacked in the delimiter stack in the ordinary REQUIRE "⊂⊃<>" DELIMITERS way. In null delimiters mode, the double quote character may be included in the macro body or macro parameter by using two double quotes:

```
DEFINE SOR = "OUTSTR(""SORRY"")";
```

The Null Delimiters mode is essentially the macro facility of ancient versions of Sail where " was the only delimiter. Programs written ancient in Sail versions will run in Null Delimiters mode. Null delimiters mode has all the rules and quirks of the prehistoric Sail macro system (the old Sail macro facility is described in [Swinehart & Sproull], Section 13). Compatibility with the ancient Sail is the only reason for Null Delimiters.

## 9.3 Macros

We will delay the discussion of macros with parameters until the next section. A macro without parameters is declared by saying:

```
DEFINE <macro_name> = <macro_body> ;
```

where <macro_name> is some legal identifier name (see page 129 for a definition of a legal identifier name). <macro_body>s can be simply a sequence of Ascii characters delimited by macro body delimiters, or they can be quite complex. Once the macro has been defined, the macro body is substituted for every subsequent appearance of the macro name. Macros may be called in this way at any point in a Sail program, except inside a Comment or a string constant.

Macro declarations may also appear virtually anywhere in a Sail program. When the word DEFINE is scanned by Sail, the scanner traps to a special production. The Define is parsed, and

the scanner returns to its regular mode as if there had been no define there at all. Thus things like

```
I ← J + 5 + DEFINE CON = c'777⊃; K12;...
```

are perfectly acceptable. However, don't put a Define in a string constant or a Comment.

## SCOPE

Macros obey block structure. Each DEFINE serves both as a declaration and an assignment of a macro body to the newly declared symbol. Two DEFINEs of the same symbol in the at the same lexical level will be flagged as an error. However, it is possible to change the macro body assigned to a macro name without redeclaring the name by using saying REDEFINE instead of DEFINE. For example,

```
BEGIN
...
  BEGIN
  ...
  DEFINE SQUAK = cOUTSTR("OUTER BLOCK")⊃;
  ...
    BEGIN
    ...
    REDEFINE SQUAK = cOUTSTR("INNER BLOCK")⊃;
    ...
    END;
  ...
  SQUAK  COMMENT Here the program types
           "INNER BLOCK";
  END;   COMMENT Here SQUAK is undefined.
       If SQUAK were included here, you'd
       get the error message
          "UNDEFINED IDENTIFIER:SQUAK";
END
```

REDEFINE of a name that has not been declared in a DEFINE will act as a DEFINE. That is, it will also declared the macro name as well as assigning a body to it.

## MACRO BODIES

A Macro Body may be

1. A sequence of Ascii characters preceded by a left macro body delimiter and followed by a right macro body delimiter.
2. An integer expression that may be evaluated at compile time.
3. A string expression that may be evaluated at compile time.
4. Concatenations of the above.

WARNING: Source file switching inside macros will not work.

## DELIMITED STRINGS

Any sequence of Ascii characters, including " may be used as a macro body if they are properly delimited. The macro body scanner keeps a count of the number of left and right delimiters seen and will terminate its scan only when it has seen the same number of each. This lets the macro body delimiters "nest" so that one may include DEFINEs inside a macro body. For example,

```
DEFINE DEF =
  cDEFINE SYM = cSYMBOL⊃; SYM⊃ ;
```

One may temporarily override the active delimiters by including a two character string before the "=" of the Define statement. For example:

```
DEFINE LES "&%" = & 0≤X<BIGGEST ∧ Y>X %;
```

The first character of the two character string becomes the left delimiter, and the second becomes the right delimiter.

## INTEGER COMPILE TIME EXPRESSIONS

Sail tries to do as much arithmetic as it can at compile time. In particular, if you have an arithmetic expression of constants, such as

```
91.504 + (3.1415↑8↑(9-7))
     % "Sail can convert strings"
```

then the whole expression will be evaluated at compile time and the resultant constant, in this case 93.9263610, will be used in your code instead of the constant expression. Runtime functions of constants will be done at compile time too, if possible. EQU and the conversion routines (CVS, CVO, etc.) will work.

When an integer compile time expression is scanned as part of a macro body, it is immediately evaluated. The integer constant which results is converted to a character string, and that character string used for the place in the macro body of the integer expression. Thus,

```
DEFINE TTYUUO = '51 LSH 27 ;
```

will cause '51 LSH 27 to be evaluated, and the resulting constant, 5502926848, will be converted to the character string 5502926848, and that character string assigned to the macro name TTYUUO.

### STRING COMPILE TIME EXPRESSIONS

If a compile time expression has the type string (constant), the macro scanner will evaluate the expression immediately. However, the string constant that results will not be converted to the character string that represents that constant, but to the character string with the same characters that the string constant had. Thus, the way to use a macro for string constants is to delimit the string constant like this:

```
DEFINE STRINCON = c"Very long
   complex string that is hard
      to type more than once"⊃ ;
```

However, the automatic conversion of string constants to character strings is helpful and indeed essential for automatic generation of identifiers:

```
DEFINE N = 1;
     COMMENT we will use this like a variable;

DEFINE GENSYM = c
  DEFINE SYM = cTEMP_⊃ & CVS(N);
     COMMENT SYM is defined to be the character
     string  TEMP_# where # is an number;

    REDEFINE N = N+1;
     COMMENT This increments N;

    SYM ⊃;
     COMMENT At the call of SYM, the character
     string is  read like program text. E.g.;

INTEGER GENSYM, GENSYM, GENSYM, GENSYM;
REAL GENSYM, GENSYM;
     COMMENT We have generated 6 identifiers with
     unique names, and declared 4 as integers,
     2 as reals;
```

To convert a macro body to a string constant, one may use CVMS. Similarly, a macro parameter is converted to a string constant by CVPS.

```
<string constant> ← CVMS (<macro name>);
<string constant> ← CVPS (<macro parameter name>)
```

A string that has the exact same characters as the macro body will be returned. For example:

```
DEFINE A = cB & C⊃;
DEFINE ABC = CVMS (A) & c & D⊃;
 COMMENT ABC now stands for the text B & C & D;
```

### HYBRID MACRO BODIES

When two delimited strings are concatenated, the result is a longer delimited string. "&" in compile time expression behaves the same way it behaves in any expression. When a compile time expression is concatenated to a delimited character string in a macro body, the result is exactly the result one would get if the delimited character string were a string constant, except that the result is a delimited character string. For example:

```
DEFINE N = 1;
DEFINE M = 2;
DEFINE SYM = CVS(N*M + N+2) & c-SQRT(N*M+1)⊃;
DEFINE SYM1 = c3-SQRT(N*M+1)⊃;
```

Here SYM is exactly the same as SYM1.

## 9.4 Macros with Parameters

One defines a macro with parameters by specifying the formal parameters in a list following the macro name:

```
DEFINE MAC (A, B) = cIF A THEN B ELSE ERR←1;⊃;
```

One calls a macro with parameters by including a list of delimited character strings that will be substituted for each occurrence of the corresponding formal in the macro body. For example,

```
COMMENT we assume that "<" and ">" are the
        parameter delimiters at this point;
MAC (<BYTES LAND (BITMASK + '2000)>, <
        BEGIN
          WWDAT ← FETCH (BYTES, ENVIRON);
          COLOR[WWDAT] ← '2000;
        END >)
```

expands to

```
IF BYTES LAND (BITMASK + '2000) THEN
        BEGIN
          WWDAT ← FETCH (BYTES, ENVIRON);
          COLOR[WWDAT] ← '2000;
        END
        ELSE ERR←1;
```

Parameter delimiters nest. Furthermore, if no delimiters are used about a parameter, nesting counts are kept of "()", "[]", and "{}" character pairs. The parameter scan will not terminate until the nesting counts of each of the three pairs is zero. One may temporarily override the active parameter delimiters by including a two character string ahead of the parameter list in the macro call:

```
MAC "∈3" (∈BYTES > '20003, ∈MATCH(BYTES)3)
```

Formal parameters may not appear in compile time expressions that are used to specify macro bodies. This is quite natural: compile time expressions must be evaluated as they are scanned, but the value of a formal parameter isn't known until later. However, if the macro body is a hybrid of expressions and delimited character strings, then formal parameters may appear in the delimited string parts.

When doing a CVMS on a macro with parameters, use only the macro name in the call; the parameters are unnecessary. The string returned will have the two character strings "λ1", "λ2", etc. (here λ stands for the Ascii character '177) where the formal parameters were in the macro body. A "λ1" will appear wherever the first formal parameter of the formal parameter list appear in the macro body, a "λ2" will appear wherever the second parameter appeared, etc. The unfortunate appearance of the Ascii character '177 in CVMS-generated strings is a product of the representation of macro bodies as strings ending in '177, '0 (which CVMS removes), having '177, n for each appearance of the nth formal parameter in the body.

## 9.5 Conditional Compilation

The compile time equivalents of the Sail IF, WHILE, FOR and CASE statements are

```
IFC <CT expr> THENC <anything> ENDC

IFC <CT expr> THENC <anything> ELSEC
      <anything> ENDC

WHILEC ⊂<CT expr>⊃ DOC ⊂<anything>⊃ ENDC

FORC <CT variable> ← <CT expr> STEPC <CT expr>
      UNTILC <CT expr> DOC ⊂<anything>⊃ ENDC

FORLC <CT variable> ← (<macro param>, ... ,
      <macro param>) DOC ⊂<anything>⊃ ENDC

CASEC <CT expr> OFC ⊂<anything>⊃, ⊂<anything>⊃,
      ... , ⊂<anything>⊃ ENDC
```

where <CT expr> is any compile time expression. <CT expr> could itself include IFCs, FORCs or whatever. <CT variable> is a macro name such as N from a define such as DEFINE N = MUMBLE; <macro param> is anything that is delimited like a macro parameter. <anything> can be anything one could want in his program at that point, including Defines and other conditional compilation statements. The usual care must be taken with nested IFCs so that the ELSECs match the desired THENCs. The "⊂" and "⊃" characters above are to stand for the current MACRO BODY DELIMITER pair.

The semantics are exactly those of the corresponding runtime statements, with one exception. When the list to a FORLC is null (i.e. it looks like "( )" ), then the <anything> is inserted in the compilation once, with the <CT variable> assigned to the null macro body.

Situations frequently occur where the false part of an IFC must have the macros in it expanded in order to delimit the false part correctly. For example,

```
DEFINE DEBUG_SELECT =
  c⊂IFC DEBNUM = 2 THENC ⊃;
DEFINE DEBUG_END =
  c⊂ELSEC OUTSTR ("DEBUG POINT") ENDC⊃;

Debug_select
    OUTSTR ("DEBUG POINT #" & CVS (DBN));
Debug_end
```

If DEBNUM is not 2, then the program must expand the macro Debug_end in order to pick up the ELSEC that terminates the false part of the conditional. The expansion is only to pick up such tokens -- the text of the false part is not sent to the scanner as the true part is. In order to avoid such expansion, one may use IFCR (the R stands for "recursive") instead of IFC.

As an added feature, when delimiters are required about an <anything> in the above (such constructs are named <delimited_anything> in the BNF), one may substitute a concatenation of constant expressions and delimited strings. This is just like a macro body, except the concatenation MUST contain at least one delimited string, thereby forcing the result of the concatenation to be a delimited string, rather than a naked expression.

As a further added feature,

```
IFC <CT expr> THENC c<anything>⊃ ELSEC
    c<anything>⊃ ENDC
```

may be substituted in FORCs, FORLCs, and WHILECs for the <anything> following DOC.

NOTE: In a WHILEC, the expression must be delimited with the appropriate macro body delimiters (hence the construct <delimited_expr> in the BNF).

## 9.6 Type Determination at Compile Time

To ascertain the type of an identifier at compile time, one may use the integer function DECLARATION (<identifier>). This returns an integer with bits turned on to represent the type of identifier. Exactly what the bits represent is a dark secret and changes periodically anyway. The best way to decode the integer returned by Declaration is to compare it to the integer returned by CHECK_TYPE (<a string of Sail declarators>). A Sail declarator is any of the reserved words used an a declaration. Furthermore, the declarators must be listed in a legal order, namely, an order that is legal in declarations (i.e. ARRAY INTEGER won't work). One may include as arguments to CHECK_TYPE the following special tokens:

| TOKEN | EFFECT |
|---|---|
| BUILT_IN | The bit that is on when a procedure is known to preserve ACs 0-'11 (except AC1 if returning a value) is returned. Sail does not clear the ACs when compiling a call on a BUILT_IN procedure. |
| LEAP_ARRAY | The bit that is on when an identifier is an item or itemvar with a declared array datum is returned (*the discussion of Leap* starts on page 83). |
| RESERVED | The bit that is on for a reserved word is returned. |
| DEFINE | The bit that indicates the identifier is a macro name is returned (note: a macro name as the argument to DECLARATION will not be expanded). |
| CONOK | The bit which says "this procedure will be evaluated at compile time if all its arguments are constant expressions" is returned. |

Examples:

```
DECLARATION (FOO) = CHECK_TYPE (INTEGER)
    This is an exact compare. Only if Foo is
    an integer variable will equality hold.

DECLARATION (A) LAND CHECK_TYPE (ARRAY)
    This is not an exact compare. If A is any
    kind of an array, the LAND will be non-zero.
```

DECLARATION (CVS) = CHECK_TYPE(EXTERNAL CONOK
OWN BUILT_IN FORWARD STRING PROCEDURE)
The equality holds. FORWARD so that you can redeclare it without complaints; OWN as a hack which saves space in the compiler.

DECLARATION (BEG) LAND CHECK_TYPE (RESERVED)
This is non-zero only if one has said
LET BEG = BEGIN. DEFINE BEG = BEGIN
will only turn the Define bit of BEG on.

NOTE: if the <identifier> of DECLARATION has not yet been declared or was declared in an inner block, then 0 is returned -- it is undeclared so it has no type.

EXPR_TYPE returns the same bits that DECLARATION does, except that the argument to EXPR_TYPE may be an expression and not just an identifier.

## 9.7 Miscellaneous Features

COMPILE TIME I/O
Compile time input is handled by the REQUIRE "<file_name>" SOURCE_FILE construct. <file_name> can be any legal file, including TTY: and MTA0: and of course disk files. (MTA does not work for TENEX.) The file will be read until the its end of file delimiter is scanned (<ctrl>Z for TTYs or <meta><ctrl><lf> at SUAI), and its text will replace the REQUIRE statement in the main file.

Compile time output is limited to typing a message on the user's teletype. To do this say REQUIRE <string_constant> MESSAGE, and the <string_constant> will appear on your teletype when the compilation hits that point in your file.

EVALDEFINE, EVALREDEFINE
The reserved word EVALDEFINE may be used in place of the word DEFINE if one would like the identifier that follows to be expanded. When one follows a DEFINE with a macro name, the macro is not expanded, but rather the macro name is declared at the current lexical level and assigned the specified macro body. EVALDEFINE gets you around that. Helps with automatic generation of macro names. EVALREDEFINE is also available.

ASSIGNC
The following compile time construct makes recursive macros easier.

ASSIGNC <name1> = <macro_body>;

<name1> must be a formal to a macro, and <macro_body> may be any macro body. Thereafter, whenever <name1> is instantiated, the body corresponding to <macro_body> is used in the expansion rather than the text passed to the formal at the macro call.

RESTRICTION: ASSIGNC may only appear in the body of the macro that <name1> is a formal of. If it appears anywhere else, the <name1> will be expanded like any good formal, and that text used in the ASSIGNC as <name1>. Unless you're being very clever, this is probably not what you want.

NOMAC
Preceding anything by the token NOMAC will inhibit the expansion of that thing should that thing turn out to be a macro.

COMPILER_BANNER
This is a predefined macro which expands to a string constant containing the text of the two-line banner which would appear at the top of the current page if a listing file were being made. This string contains the date, time, name and page of the source file, the value of all compiler switches, the name of the outer block, and the name of the current block. Thus you can automatically include the date of compilation in a program by using COMPILER_BANNER[n TO m] for appropriate n and m. Try REQUIRE COMPILER_BANNER MESSAGE; or look at a listing for the exact format.

## 9.8 Hints

The following is a set of hints and aids in debugging programs with macros. Unless otherwise stated array brackets "[]" are the macro body delimiters.

IFC and friends will not trigger at the point of macro definition, in a macro actual parameter list, or inside a string constant.

```
DEFINE FOO = [IFC A THENC B ELSEC D ENDC];
      which is not the same as
DEFINE FOO = IFC A THENC [B] ELSEC [D] ENDC;
      which is the same as
IFC A THENC DEFINE FOO = [B]
     ELSEC DEFINE FOO = [D] ENDC;

DEFINE BAZ (A) = [OUTSTR ("A");];
BAZ (IFC B THENC C ELSEC D ENDC)
   will result in the following string typed
   on your terminal:
IFC B THENC C ELSEC D ENDC

STRING A;
A←"IFC WILL NOT TRIGGER HERE";
```

Macros will not be expanded in strings, but macro formal parameters will be expanded when they occur in strings within macro bodies as seen in the second example above.

```
DEFINE FOO = [BAZ];
OUTSTR ("FOO");
```

which will type out the string FOO on your terminal rather than BAZ.

Caution should be employed when using letters (specifically ⊂⊃) as delimiters. This may lead to problems when defining macros within macros.

```
DEFINE MAC(A) "⊂⊃" = ⊂REDEFINE FOO =⊂A⊃;⊃;
```

Inside the macro body of MAC, A will not be recognized as a formal since the scanner has scanned ⊂A⊃ as an identifier by virtue of ⊂⊃ being internally represented as letters so that they could be defined to mean BEGIN and END respectively (also ⊂ as COMMENT). More justification for this feature is seen by the following example:

```
DEFINE MAC(ABC) "AC" = A V←ABC; C;
```

We want ABC in the text to be the parameter and not B if we were to ignore the macro delimiters.

When scanning lists of actual parameters, macros are not expanded.

```
DEFINE FOO = [A,B];
   MAC (FOO) will not have the result MAC(A,B). However,
DEFINE FOO = [(A, B)];
   followed by MAC FOO will have the same effect as
   MAC (A, B).
```

The same reasoning holds for parameter lists to FORLC.

```
DEFINE FOO = [A, B, C];
FORLC I = (FOO) DOC [OUTSTR ("I");] ENDC
   will result in FOO typed out on your terminal.

DEFINE FOO = [(A, B, C)];
FORLC I = FOO DOC [OUTSTR ("I");] ENDC
   will have the desired result ABC typed out.
```

In order to take advantage of the nestable character feature in the parameters to a macro call, one must be in REQUIRE DELIMITERS mode. Otherwise scanning will break upon seeing a comma or a right parenthesis.

```
BEGIN
  DEFINE FOO(A) = "A";
  INTEGER ARRAY ABC[1:10, 1:10];
  FOO (ABC[1, 2])←3;
END;
```

This is identical to:

```
BEGIN
  INTEGER ARRAY ABC[1:10, 1:10];
  ABC[1←3; Comment illegal;
END;
```

However, if the original program had included a REQUIRE DELIMITERS statement prior to the macro call, as below, then the desired effect would have resulted - i.e., ABC[1, 2]←3 .

```
BEGIN
  REQUIRE "{}?$" DELIMITERS;
  DEFINE FOO (A) = {A};
  INTEGER ARRAY ABC[1:10, 1:10];
  FOO (ABC[1, 2])←3;
END;
```

# SECTION 10

# RECORD STRUCTURES

## 10.1 Introduction

Record structures are new to Sail. They provide a means by which a number of closely related variables may be allocated and manipulated as a unit, without the overhead or limitations associated with using parallel arrays and without the restriction that the variables all be of the same data type. In the current implementation, each record is an instance of a user-defined record class, which serves as a template describing the various fields of the record. Internally, records are small blocks of storage which contain space for the various fields and a pointer to a class descriptor record. Fields are allocated one per word and are accessed by constant indexing off the record pointer. Deallocation is performed automatically by a garbage collector or manually through explicit calls to a deallocation procedure.

## 10.2 Declaration Syntax

```
<record_class_declaration>
     ::= RECORD_CLASS <class_id> (
          <field_declarations> )
     ::= RECORD_CLASS <class_id> (
          <field_declarations> ) [ <handler> ]

<record_pointer_declaration>
     ::= RECORD_POINTER ( <classid_list>
          ) <id_list>
     ::= RECORD_POINTER ( ANY_CLASS
          ) <id_list>
```

## 10.3 Declaration Semantics

The <field_declarations> have the same form as the <formal_param_decl> of a procedure, except that the words VALUE and REFERENCE should not be used, and default values are ignored. Each record class declaration is compiled into a record descriptor (which is a record of constant record class $CLASS) and is used by the runtime system for allocation, deallocation, garbage collection, etc. At runtime record pointer variables contain either the value NULL_RECORD (internally, zero) or else a pointer to a record. The <classid list> is used to make a compile-time check on assignments and field references. The pseudo-class ANY_CLASS matches all classes, and effectively disables this compile-time check.

For instance,

```
RECORD_CLASS VECTOR (REAL X, Y, Z);
RECORD_CLASS CELL
   (RECORD_POINTER (ANY_CLASS) CAR, CDR);
RECORD_CLASS TABLEAU
   (REAL ARRAY A, B, C; INTEGER N, M);
RECORD_CLASS FOO (LIST L; ITEMVAR A);

RECORD_POINTER (VECTOR) V1,V2;
RECORD_POINTER (VECTOR, TABLEAU) T1,T2;
RECORD_POINTER (ANY_CLASS) R;

RECORD_POINTER (FOO, BAR) FB1, FB2;
RECORD_POINTER (FOO) FB3;
RECORD_POINTER (CELL) C;
RECORD_POINTER (ANY_CLASS) RP;

COMMENT the following are all ok syntactically;
C ← NEW_RECORD (CELL);
RP ← C;
FB2 ← NEW_RECORD (FOO);
FB1 ← FB3;
FB3 ← RP; COMMENT This is probably a runtime bug
        since RP will contain a cell record.  Sail
        won't catch it, however;
CELL:CAR[RP] ← FB1;
CELL:CAR[RP] ← FB1;

COMMENT The compiler will complain about these: ;
FB1 ← C;
FB3 ← NEW_RECORD (CELL);
RP ← CELL:CAR[FB3];
```

NO runtime class information is kept with the record pointer variables, and no runtime class

checks are made on record assignment or field access. Record pointer variables are allocated quantities, and should not appear inside SIMPLE procedures. They resemble lists in that they are not given any special value upon block entry and they are set to a null value (NULL_RECORD) when the block in which they are declared is exited. (This is so that any records referred to only in that block can be reclaimed by the garbage collector.)

Record pointers are regular Sail data types, just like integers or strings; record pointer procedures, arrays, and items all work in the normal way. As indicated earlier, the constant NULL_RECORD produces a null reference.

## 10.4 Allocation

Records are allocated by

```
NEW_RECORD (<classid>)
```

which returns a new record of the specified class. All fields of the new record are set to the null or zero value for that field; i.e., real and integer fields will be set to 0, itemvar fields to ANY, lists to NIL, etc. Note that entry into a block with local record pointer variables does NOT cause records to be allocated and assigned to those variables.

## 10.5 Fields

Record fields are referenced by

```
<classid> : <fieldid> [ <record pointer expression> ]
```

and may be used wherever an array element may be used. For example

```
RECORD_POINTER (VECTOR) V;
RECORD_POINTER (CELL) C;
RECORD_POINTER (FOO) F;

VECTOR:X[V] ← VECTOR:Y[V];
CELL:CAR[C ← NEW_RECORD (CELL)] ← V;
VECTOR:Z[V] ← VECTOR:X[CELL:CAR[C]];
SUBLIS ← FOO:L[F][1 TO 3];
```

If the <record pointer expression> gives a null record, then a runtime error message will be generated. This is the only runtime check that is made at present. I.e., no runtime checks are made to verify that the <classid> in the field statement matches the class of the record whose field is being extracted.

An array field may be used as an array name, as in

```
RECORD_POINTER (TABLEAU) T;

TABLEAU:A[T][I,J] ← 2.5;
```

provided that a valid array descriptor has been stored into the field. Unfortunately, Sail does not provide any clean way to do this. One unclean way is

```
EXTERNAL INTEGER PROCEDURE ARMAK
    (INTEGER LB, UB, *DIMS);
COMMENT returns address of first data word of new
    array. For String arrays set *DIMS to -1,,n
    For higher dimensions declare with more LB, UB pairs.

EXTERNAL PROCEDURE ARYEL (INTEGER ARR);
COMMENT deallocates an array. ARR is the address of
    the first data word;

RECORD_CLASS FUBAR (INTEGER ARRAY A);
RECORD_POINTER (FUBAR) FB;

MEMORY[LOCATION (FUBAR:A[FB])] ← ARMAK (1, 100, 1);
ARYEL (MEMORY[LOCATION (FUBAR:A[FB])]);
```

(Warning: the above advice is primarily intended for hackers. No promises are made that it will always work, although this particular trick is unlikely to be made obsolete in the forseeable future.)

## 10.6 Garbage Collection

The Sail record service routines allocate records as small blocks from larger buffers of free storage obtained from the normal Sail free storage system. (The format of these records will be discussed in a later section.) From time to time a garbage collector is called to reclaim the storage for records which are no longer accessible by the user's program (i.e., no variables or accessible records point to them).

The garbage collector may be called explicitly from Sail programs as external procedure $RECGC, and automatic invocation of the garbage collection may be inhibited by setting user table entry RGCOFF to TRUE. (In this case, Sail will just keep allocating more space, with nothing being reclaimed until RGCOFF is set back to FALSE or $RECGC is called explicitly). In addition, Sail provides a number of hooks that allow a user to control the automatic invocation of the garbage collector. These are discussed later.

## 10.7 Internal Representations

Each record has the following form:

```
-1:  <ptrs to ring of all records of class>
 0   <garbage collector ptr>,,<ptr to class descriptor>
+1:    <first field>
 :      :
+n:    <last field>
```

Record pointer variables point at word 0 of such records. A String field contains the address of word2 of a string descriptor, like the string was a REFERENCE parameter to a procedure. The string descriptors are also dynamically allocated.

The predefined record class $CLASS defines all record classes, and is itself a record of class $CLASS.

```
RECORD_CLASS $CLASS
   (INTEGER RECRNG, HNDLER, RECSIZ;
    INTEGER ARRAY TYPARR; STRING ARRAY TXTARR);
```

RECRNG is a ring (bidirectional linked list) of all records of the particular class.

HNDLER is a pointer to the handler procedure for the class (default $REC$).

RECSIZ is the number of fields in the class.

TYPARR is an array of field descriptors for each field of the class.

TXTARR is an array of field names for the class.

The normal value for the handler procedure is $REC$, which is the standard procedure for such functions as allocation, deallocation, etc.

TYPARR and TXTARR are indexed [0:RECSIZ]. TXTARR[0] is the name of the record class. TYPARR[0] contains type bits for the record class.

Example:

```
RECORD_CLASS FOO (LIST L; ITEMVAR A);
```

The record class descriptor for FOO contain:

```
FOO-1:  <ptrs for ring of all records of $CLASS>
FOO:    <ptr to $CLASS>
FOO+1:  <ptrs for ring of all records of class FOO;
           initialized to <FOO+2,,FOO+2> >.
FOO+2:  <ptr to handler procedure $REC$>
FOO+3:  2
FOO+4   <ptr to TYPARR>
FOO+5:  <ptr to TXTARR>
```

The fields of FOO are:

```
$CLASS:RECRNG[FOO] = <initialized to null ring,
                      i.e., xwd(loc(FOO)+2,loc(FOO)+2)>
$CLASS:HNDLER[FOO] = $REC$
$CLASS:RECSIZ[FOO] = 2
$CLASS:TXTARR[FOO] [0] = "FOO"
$CLASS:TXTARR[FOO] [1] = "L"
$CLASS:TXTARR[FOO] [2] = "A"
$CLASS:TYPARR[FOO] [0] = <bits for garbage collector>
$CLASS:TYPARR[FOO] [1] = <descriptor for LIST>
$CLASS:TYPARR[FOO] [2] = <descriptor for ITEMVAR>
```

## 10.8 Handler Procedures

Sail uses a single runtime routine $RECFN (OP, REC) to handle such system functions as allocation, deallocation, etc. The code compiled for r ← NEW_RECORD (foo) is

```
PUSH    P,[1]
PUSH    P,[foo]
PUSHJ   P,$RECFN
MOVEM   1,r
```

$RECFN performs some type checking and then jumps to the handler procedure for the class. The normal value for this handler procedure is $REC$. It is possible to substitute another handler procedure for a given class of records

by including the procedure name in brackets after the record class declaration. The handler must have the form

```
RECORD_POINTER (ANY_CLASS) PROCEDURE <procid>
   (INTEGER OP; RECORD_POINTER (ANY_CLASS) R);
```

Here OP will be a small integer saying what is to be done. The current assignments for OP are:

| value | meaning |
|---|---|
| 0 | invalid |
| 1 | allocate a new record of record class R |
| 2 | not used |
| 3 | not used |
| 4 | mark all fields of record R |
| 5 | delete all space for record R |

At SUAI, macro definitions for these functions may be found in the file SYS:RECORD.DEF, which also includes EXTERNAL declarations for $CLASS, $REC$, and $RECFN.

$REC$ (1, R) allocates a record of the record class specified by R, which must be a record of class $CLASS. All fields (except string) are initialized to zero. String fields are initialized to a pointer to a string descriptor with length zero (null string).

$REC$ (4, R) is used by the garbage collector to mark all record fields of R.

$REC$ (5, R) deallocates record R, and deallocates all string and array fields of record R. Care must be exercised to prevent multiple pointers to string and array fields; i.e., DO NOT store the location of an array in fields of two different records unless extreme caution is taken to handle deletion. This can be accomplished through user handler procedures which zero array fields (without actually deleting the arrays) prior to the call on $REC$ (5, R).

NOTE: When an alternate handler procedure is supplied it must perform all the necessary functions. One good way to do this is to test for those OPs performed by the alternate handler and call $REC$ for the others. If $REC$ is used to allocate space for the record then it should also be used to release the space. These points are illustrated by the following example:

```
FORWARD RECORD_POINTER (ANY_CLASS) PROCEDURE
   FOOH (INTEGER OP;
         RECORD_POINTER (ANY_CLASS) R);
RECORD_CLASS FOO (ITEMVAR IV) [FOOH];
RECORD_POINTER (ANY_CLASS) PROCEDURE FOOH
   (INTEGER OP; RECORD_POINTER (ANY_CLASS) R);
 BEGIN
 PRINT("CALLING FOOH.  OP = ", OP);
 IF OP = 1 THEN
    BEGIN
    RECORD_POINTER (FOO) F;
    F ← $REC$ (1,R);
    FOO:IV[F] ← NEW;
    RETURN (F);
    END
 ELSE IF OP = 5 THEN
    DELETE (FOO:IV[R]);
 RETURN ($REC$ (OP, R));
 END;
```

## 10.9 More about Garbage Collection

The information used by the system to decide when to call $RECGC on its own is accessible through the global array $SPCAR. In general, $SPCAR[n] points at a descriptor block used to control the allocation of small blocks of n words. This descriptor includes the following fields:

| | |
|---|---|
| BLKSIZ | number of words per block in this space |
| TRIGGER | a counter controlling time of garbage collection |
| TGRMIN | described below |
| TUNUSED | number of unused blocks on the free list |
| TINUSE | total number of blocks in use for this space |
| CULPRIT | the number of times this space has caused collection |

The appropriate macro definitions for access to these fields may be found in the source file <SUAI>SYS:RECORD.DEF. The decision to invoke the garbage collector is made as part of the block allocation procedure, which works roughly as follows:

```
INTEGER spc,size;
size ← $CLASS.RECSIZ[classid]+2;
IF size>16 THEN return a CORGET block;
spc ← $SPCAR[size];
L1:
IF (MEMORY[spc+TRIGGER]
        ← MEMORY[spc+TRIGGER]-1) <0
   THEN BEGIN
  IF ¬MEMORY[GOGTAB+RGCOFF] THEN BEGIN
    MEMORY[spc+CULPRIT] ← MEMORY[spc+CULPRIT]+1;
    $RECGC;
    GO TO L1;
END END;
<allocate the block from space spc,
 update counters, etc.>
```

Once $RECGC has returned all unused records to the free lists associated with their respective block sizes, it must adjust the trigger levels in the various spaces. To do this, it first looks to see if the user has specified the location of an adjustment procedure in TGRADJ(USER). If this cell is non-zero then $RECGC calls that procedure (which must have no parameters). Otherwise it calls a default system procedure that works roughly like this:

```
<set all TRIGGER levels to -1>
FOR size ← 3 STEP 1 UNTIL 16 DO BEGIN
   spc ← $SPCAR[size];
   IF MEMORY[spc+TRIGGER]<0 THEN BEGIN
      t←MEMORY[spc+TINUSE]·RGCRHO(USER);
      t←MAX(t, MEMORY[spc+TUNUSED],
            MEMORY[spc+TGRMIN]);
END END;
```

RGCRHO(USER) is a real number currently initialized by the system to 0.33. Thus the behavior of Sail's automatic garbage collection system may be modified by

Setting RGCOFF(USER).
Supplying a procedure in TGRADJ(USER).
Modifying RGCRHO(USER).
Modifying the TGRMIN entries in the space descriptors.

One word of caution: User procedures that set trigger levels must set the trigger level of the space that caused garbage collection to some positive value. If not then a runtime error message will be generated.

Look at the file ⊂SUAI⊃RECAUX.SAI[CSP,SYS], which contains a number of useful examples and auxilliary functions.

# SECTION 11

# TENEX ROUTINES

## 11.1 Introduction

This section describes routines which interface Sail with the TENEX operating system. Routines for file input/output, terminal handling, and miscellaneous system calls are described here. For TENEX-specific details of other routines (such as interrupts) consult the appropriate chapter.

## 11.2 TOPS-10 Style Input/Output

"Standard" Sail programs written using TOPS-10 I/O routines such as OPEN, LOOKUP, etc., will run under TENEX with little or no conversion necessary. The TENEX Sail routines simulate most of the effects of the TOPS-10 I/O calls without using the PA-1050 emulator.

In TENEX Sail the non-zero values of error flags returned by routines such as LOOKUP are ERSTR JSYS error numbers. The interpretation of zero/nonzero is the same as with the TOPS-10 I/O routines, but the specific nonzero values are probably different.

Here are the TOPS-10 I/O routines and the differences, if any, under TENEX.

ARRYIN TENEX dump mode implies a single DUMPI JSYS.

ARRYOUT similar to ARRYIN.

CLOSE The close inhibit bits have no effect.

CLOSIN same as CLOSE.

CLOSO same as CLOSE.

ENTER no differences.

GETCHAN In TOPS-10, GETCHAN returns the number of a channel for which no OPEN is currently in effect. Thus successive GETCHANs without intervening OPENs will return the same channel number. In TENEX Sail, GETCHAN returns the number of a channel for which no OPEN or GETCHAN is currently in effect; thus successive GETCHANs will return different channel numbers.

GETSTS not available; see GDSTS, GTSTS.

INOUT not available.

INPUT assumes 200 characters maximum if no length variable has been associated with the channel.

INTIN no differences.

LINOUT no differences.

LOOKUP no differences.

MTAPE Options "I" and NULL are not available.

OPEN MODE is mostly ignored (exception: dump mode on a dectape ignores the directory). The number of input and output buffers serves only to indicate whether reading or writing is desired.

OUT no differences.

REALIN no differences.

RELEASE The close inhibit bits have no effect.

RENAME Changing the protection does not work. See GTFDB and CHFDB.

SETPL The routines CHARIN and SINI do not update the variables associated with the channel by SETPL.

SETSTS not available; see SDSTS, STSTS.

STDBRK no differences.

TMPIN not available.

TMPOUT not available.

USETI works only on those devices where the SFPTR JSYS works. On a dectape the MTOPR JSYS is used, and may not produce the same results as on a TOPS-10 system. USETI takes effect

immediately (the nondeterminancy of the standard TOPS-10 (not SUAI) USETI is not simulated). Equivalent to SFPTR (chan, (N-1)*'200);

USETO same as USETI. TENEX has only one file pointer, so in fact USETI and USETO are EXACTLY the same function.

WORDIN no differences.

WORDOUT no differences.

MAGTAPE I/O
The user is warned that there are serious limitations in TENEX regarding magtapes. While TENEX is supposed to have device-independent I/O, the magtape code in TENEX (as of v. 1-31) is minimal, allowing only dump mode transfers. Further, end of file markers must be written explicitly, and it is sometimes necessary to do an MTOPR operation 0 to reset the magtape status bits.

TENEX Sail has been designed to handle some of these things in a way that makes features available on a standard TOPS-10 system available in a transparent way. For example, string input and output functions work, with Sail assuming 128-word records on the tape. ARRYIN and ARRYOUT cause the DUMPI and DUMPO JSYSes to be executed for the specified word counts. TENEX Sail does not actually open tapes for write until a write operation is requested. A CLOSF or CFILE on a tape will write two EOF'S (MTAPE (ch, "E")) and backspace over one of them, if and only if the file has been opened. Do not rewind a tape unless it has been closed. The user who wants to write magtape code for operations other than the above is hereby warned that the TENEX magtape code is fraught with peril. TENEX Sail certainly allows full access to TENEX in this regard, however.

## 11.3 TENEX Style Input/Output

The following functions satisfy most Sail and TENEX needs:

ARRYIN Read in an array (36-bit words)

ARRYOUT Write an array

CFILE Release a file

CPRINT Write a string

INPUT Read in a string

JFNS Read file name

OPENFILE Obtain a file

OUT Write a string

SETINPUT Set parameters for input

OBTAINING ACCESS
The main procedure for obtaining access to files is OPENFILE. In terms of JSYSes, OPENFILE does a GTJFN and OPENF. Additional routines provide support to OPENFILE, including SETINPUT, INDEXFILE, and CFILE.

DATA TRANSFER
The TENEX routines for transferring data are generally the same as the TOPS-10 routines. One improvement in TENEX Sail is that characters and words can be mixed in reading or writing to a file, provided the file is on the disk. Such I/O is called "data mixed I/O".

The following interpretation is given to data mixed I/O. There is one logical character pointer into the file. When a character is read or written the routines access the byte designated by the pointer and then increment the pointer. There is only one pointer for both input and output. When a word is read or written, the next full word in the file is accessed. Accessing a word advances the character pointer to the next full word in the file, where five 7-bit ASCII characters occupy one 36-bit word. If a read passes the end of the file then the EOF variable (specified by SETINPUT or OPEN) and the external integer _SKIP_ are set to -1. If a write passes the end of file then the end of file is advanced.

RANDOM I/O
The routines RCHPTR, SCHPTR, RWDPTR, and SWDPTR give access to the file pointer. USETI and USETO are equivalent to SWDPTR (chan, (N-1)*'200);.

ERROR HANDLING
When errors occur the runtime routines will sometimes trap the errors themselves. This practice is held to a minimum since the error itself may be information that the user is interested in seeing. Usually the routines (as marked) put the TENEX error code in _SKIP_, which may be examined by the program. The TENEX error numbers do not always make good sense, but for the cases that they do the ERSTR routine will print out on the terminal the message associated with a given error number.

DIRECT DSK OPERATIONS
The routines DSKIN and DSKOUT do direct DSK operations in TENEX Sail, using the DSKOP JSYS. These routines relate only to the IMSSS version of TENEX-Sail.

### ASND, RELD

SUCCESS ← ASND (DEVICE_DESCRIPTOR);
SUCCESS ← RELD (DEVICE_DESCRIPTOR)

DEVICE_DESCRIPTOR (in the TENEX sense) is assigned to or deassigned from the job. If DEVICE_DESCRIPTOR is -1 when calling RELD then all devices assigned to the job are deassigned. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

### BKJFN

BKJFN (CHAN)

Does the BKJFN JSYS on CHAN. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred. This function is escape from Sail.

### CFILE

SUCCESS ← CFILE (CHAN)

This routine closes the file (CLOSF) and releases the CHAN (RLJFN). This is the ordinary way to dispense with a file. CFILE returns TRUE if CHAN is legal and released; it returns FALSE otherwise.

### CHARIN

CHAR ← CHARIN (CHAN)

The next character from CHAN is returned. Zero is returned if the file is at the end.

### CHAROUT

CHAROUT (CHAN, CHAR)

The single character CHAR is written to CHAN.

### CHFDB

CHFDB (CHAN, DISPLACEMENT,
MASK, CHANGED_BITS)

This routine performs the CHFDB JSYS on CHAN, with DISPLACEMENT, MASK, and CHANGED_BITS as described in the JSYS manual.

### CLOSF

CLOSF (CHAN)

This routine does a CLOSF on CHAN. CHAN is not released. If the device is a magtape open for output then 2 file marks are written and a backspace is performed. This writes a standard end-of-file on the tape.

### CVJFN

REAL_JFN ← CVJFN (CHAN)

The full TENEX JFN (including flags in the left half) corresponding to Sail channel CHAN is returned. Only a hacker will ever need this.

### DELF

DELF (CHAN)

The file on CHAN (which must NOT be open), is

deleted. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

### DELNF

DELETED ← DELNF (CHAN, KEPT)

This routine deletes all but KEPT versions of the file on CHAN, which must have had a CLOSF done on it first. If KEPT=0 then all versions of the file are deleted. If KEPT=1 then all versions except the most recent are deleted. The number of files actually deleted is returned as the value of DELNF.

### DEVST, STDEV

"DEVICE_NAME" ← DEVST (DEVICE_DESIGNATOR);
DEVICE_DESIGNATOR ← STDEV ("DEVICE_NAME")

These routines convert between string DEVICE_NAMEs (such as "DTA0") and TENEX DEVICE_DESIGNATORs. TENEX does not believe that lower case letters are equivalent to upper case letters in STDEV. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

### DEVTYPE

DEVICE_TYPE ← DEVTYPE (CHAN)

The DVCHR JSYS is used to return the device type of the device open on CHAN.

### DSKIN, DSKOUT

DSKIN (MODULE, RECNO, COUNT, @LOC);
DSKOUT (MODULE, RECNO, COUNT, @LOC)

[IMSSS only.] These routines do direct DSK I/O. MODULEs 4-7 are legal for everyone; other modules require enabled status. The routines transfer COUNT (≤'1000) words, starting at location LOC in memory and at record RECNO in MODULE. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred. WARNING: No bounds checking is performed to see if the LOC is a legal Sail array.

### DVCHR

DEVICE_CHAR ← DVCHR (CHAN, @AC1, @AC3)

The DEVCHR JSYS is performed. The flags from AC2 are returned as the value of the call, and AC1 and AC3 get the contents of ac's 1 and 3.

### ERSTR

ERSTR (ERRNO, FORK)

Using the ERSTR JSYS, this routine types on the console the TENEX error string associated with ERRNO for fork FORK ('400000 for the current fork). Parameters (in the sense of the ERSTR JSYS) are expanded. Types ERSTR: UNDEFINED ERROR NUMBER (and sets _SKIP_ to -1) if something is wrong with ERRNO or FORK.

### GDSTS, SDSTS

STATUS ← GDSTS (CHAN, @WORD_COUNT);
SDSTS (CHAN, NEW_STATUS)

The status of the device on CHAN is returned or changed. For GDSTS, @WORD_COUNT is set to the contents of AC3.

Remark: some magtape statuses (such as EOF) are set by MTOPR and not by SDSTS. Ordinarily the Sail runtime system takes care of this, but it is worth mentioning since so many users have run into this poorly documented fact about TENEX.

### GNJFN

MORE_FILES ← GNJFN (CHAN)

Does the GNJFN JSYS. A file that is open cannot have GNJFN applied to it. INDEXFILE should normally be used instead of GNJFN. An exception is if files are being indexed without actually being opened (i.e., without an OPENF JSYS), which is a sensible way of performing operations such as counting the number of files in a group.

## GTFDB

GTFDB (CHAN, @BUF)

The entire FDB of CHAN is read into the array BUF. No bounds checking is performed, so BUF should be at least '25 words.

## GTJFN

CHAN ← GTJFN ("NAME", FLAGS)

Does a GTJFN. If NAME is non-null then it is used, otherwise the terminal is queried for a filename. Any error code is returned in _SKIP_. The Sail channel number obtained is returned as the value of GTJFN.

The following values for FLAGS will be translated by Sail before doing the JSYS:

| value | translated to |
|---|---|
| 0 | '100001000000 (ordinary input) |
| 1 | '600001000000 (ordinary output) |

*Other values are taken literally.*

Ordinarily OPENFILE will be used rather than GTJFN. The routines GTJFN, OPENF, GNJFN, CLOSF, RLJFN, and DVCHR are all in the category of being included only for completeness; they are not necessary in most programs.

## GTJFNL

CHAN ← GTJFNL ("ORIGSTR", FLAGS, JFN_JFN,
"DEV", "DIR", "NAM", "EXT",
"PROT", "ACCOUNT", DESIRED_JFN)

Does the long form of the GTJFN JSYS (and does not do an OPENF). The arguments are put into the accumulators and locations in the table accepted by the long form of the GTJFN JSYS. These arguments are given below, where "AC X" means an accumulator and "E+X" means in the Xth address of the table.

| Argument | Where placed | What |
|---|---|---|
| "ORIGSTR" | AC 2 | Partial or complete string |
| FLAGS | E+0 | Flags to GTJFN |
| JFN_JFN | E+1 | xwd input JFN, output JFN |
| "DEV" | E+2 | device |
| "DIR" | E+3 | directory |
| "NAME" | E+4 | name |
| "EXT" | E+5 | extension |
| "PROT" | E+6 | protection |
| "ACCOUNT" | E+7 | account |
| DESIRED_JFN | E+'10 | desired JFN if B11 on |

## GTSTS, STSTS

STATUS ← GTSTS (CHAN);
STSTS (CHAN, NEW_STATUS)

These routines examine and change the file status using the JSYSes. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

WARNING: The results of GTSTS are not necessarily appropriate for determining end-of-file if the file is being page-mapped by Sail. Look at the EOF variable instead. See SETINPUT.

## INDEXFILE

ANOTHER ← INDEXFILE (CHAN)

If CHAN was opened with the "*" option by OPENFILE then INDEXFILE will try to get the next file in the "*" group. INDEXFILE returns TRUE as long as another file can be found on CHAN. Example:

```
JFN ← OPENFILE ("<JONES>*.SAI,*", "RO*");
   COMMENT Read all of Jones's Sail programs;
SETINPUT (JFN, 200, 0, EOF);

DO BEGIN "INDEX"
   DO BEGIN "READ FILE"
      STRING S;
      S ← INPUT (JFN, BREAK_TABLE);
      COMMENT process ...;
     END "READ FILE" UNTIL EOF;
  END "INDEX" UNTIL NOT INDEXFILE (JFN);
```

The "*" option takes the place of reading the MFD and UFD on a TOPS-10 system. INDEXFILE clears the EOF, LINNUM, SOSNUM, and PAGNUM variables associated with CHAN if these have been set by SETINPUT and SETPL.

### JFNS

"NAME" ← JFNS (CHAN, FLAGS)

The name of the file associated with CHAN is returned. FLAGS are for accumulator 3 as described in the JSYS manual. Zero is a reasonable value for FLAGS.

### JFNSL

"NAME" ← JFNSL (CHAN, FLAGS, LHFLAGS)

(This routine corrects a deficiency in the JFNS function.) The name of the file associated with CHAN is returned, using FLAGS for accumulator 3 and putting LHFLAGS into the left half of accumulator 2 as described in the JSYS manual. If LHFLAGS is -1 then the value returned by GTJFN is used.

### MTOPR

MTOPR (CHAN, FUNCTION, VALUE)

The MTOPR JSYS is executed with FUNCTION placed into AC2 and VALUE into AC3. The TOPS-10 style MTAPE function may be more comfortable. [(Stupid!) IMSSS and SUMEX: skip to end of tape does not work.]

### OPENF

OPENF (CHAN, FLAGS)

Does the OPENF JSYS on CHAN with FLAGS as the contents of accumulator 2. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred. The following values for FLAGS will be translated by Sail before setting AC2:

| value | translated to | |
|---|---|---|
| 0 | '070000200000 | (input characters) |
| 1 | '070000100000 | (output characters) |
| 2 | '440000200000 | (input words) |
| 3 | '440000100000 | (output words) |
| 4 | '447400200000 | (dump read) |
| 5 | '447400100000 | (dump write) |

Values 6-10 are reserved for expansion; other values are taken literally.

Best results are obtained by opening a TTY in 7-bit mode, the DSK or DTA in 36-bit mode, and a magtape in 36-bit dump mode.

### OPENFILE

CHAN ← OPENFILE ("NAME", "OPTIONS")

NAME is the name of the file to be opened. If it is null then OPENFILE gets the filename from the terminal using TENEX filename recognition. CHAN, the value returned by OPENFILE, is a Sail channel number. This is not necessarily the same as the TENEX JFN (see CVJFN). All TENEX Sail functions (except SETCHAN) require Sail channel numbers for arguments. OPTIONS is one or more characters specifying the kind of access desired. The legal characters are

Read or write:
- R read
- W write
- A append

Version numbering, old-new:
- O old file
- N new file
- T temporary file
- * index with INDEXFILE routine

Independent bits to be set:
- C require confirmation
- D ignore deleted bit
- H "thawed" access

Error handling:
- E return errors to user in the external integer _SKIP_. TENEX error codes are used. (CHAN will be released in this case.)

If an error occurs and mode "E" was not specified then OPENFILE gives an error message and attempts to obtain a file name from the

terminal. If an error occurs when "E" was specified then OPENFILE will return -1 for CHAN and the TENEX error code will be put into _SKIP_.

Examples:

```
COMMENT get a filename from the terminal
   and write the file;
BEGIN
INTEGER JFN;
OUTSTR (CRLF & "FILE NAME* ");
JFN ← OPENFILE (NULL, "WC");
   COMMENT write, confirm name;
CPRINT (JFN, "text
");
CFILE (JFN);        COMMENT close the file;
END;

COMMENT read a known file;
BEGIN
STRING S;
INTEGER JFN, BRCHAR, EOF;
SETBREAK (1, '12, '15&'14, "IN");
JFN ← OPENFILE ("<JONES>SECRET.DATA", "RC0");
SETINPUT (JFN, 200, BRCHAR, EOF);
DO  BEGIN
   S ← INPUT (JFN, 1);
   END UNTIL EOF;
CFILE (JFN);
END;
```

Wizards: The OPENF is for 36-bit transfers; except that TTY, LPT, and a device for which a 36-bit OPENF fails get 7-bit mode.

### RCHPTR, SCHPTR

PTR ← RCHPTR (CHAN);
SCHPTR (CHAN, NEWPTR)

The number of the byte which will be accessed next by character I/O is returned or set. The first character of a file is character number 0. If NEWPTR=-1 for SCHPTR then the pointer is set to end of file. Setting the pointer beyond end of file will change the length of the file if it is being written. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

### RFBSZ

BYTE_SIZE ← RFBSZ (CHAN)

The byte_size of the file open on CHAN is returned. This function is escape from Sail.

### RFPTR, SFPTR

PTR ← RFPTR (CHAN);
SFPTR (CHAN, NEWPTR)

These routines perform JSYSes and are escape from Sail. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

### RLJFN

RLJFN (CHAN)

This routine does the RLJFN JSYS.

### RNAMF

SUCCESS ← *RNAMF (EXISTINGCHAN, NEWCHAN)*

The RNAMF JSYS is performed, renaming the file on EXISTINGCHAN to the name of the (vestigial) file on NEWCHAN. It is necessary that CLOSF(EXISTINGCHAN) be done before RNAMF and that OPENF be done afterwards. The TOPS-10 style RENAME is sometimes more convenient to use than RNAMF, since RENAME performs the GTJFN and OPENFs necessary for the renaming operation. However, the actual JFN associated with CHAN is changed by RENAME.

### RWDPTR, SWDPTR

PTR ← RWDPTR (CHAN);
SWDPTR (CHAN, NEWPTR)

The number of the word which will be accessed next by word I/O is returned or set. The first word of a file is word number 0. If NEWPTR=-1 for SWDPTR then the pointer is set to end of

file. Setting the pointer beyond end of file will change the length of the file if it is being written.

### SETCHAN

CHAN ← SETCHAN (REAL_JFN,
GTJFN_FLAGS, OPENF_FLAGS)

This function is liberation from Sail I/O. It is provided for doing Sail I/O on a JFN that is obtained from some means other than the Sail file-opening routines -- for example, a JFN passed from a superior fork.

REAL_JFN is a 36-bit JFN (or JFN substitute, such as a Teletype number), GTJFN_FLAGS and OPENF_FLAGS are the flags that should be recorded describing how the GTJFN and OPENF were accomplished. REAL_JFN need not be open. The value returned by SETCHAN is the Sail channel number which should be used for subsequent Sail I/O. SETCHAN is the only function in TENEX Sail that takes an actual JFN as an argument.

### SETINPUT

SETINPUT (CHAN, @COUNT, @BRCHAR, @EOF)

This function relates the COUNT, BRCHAR, and EOF variables to channel CHAN in the same way that OPEN does. The INPUT function (page 39) uses 200 for the default value of COUNT if no location has been associated with CHAN.

All I/O transfer routines also set _SKIP_ to indicate end-of-file and I/O errors. For example, on return from INPUT _SKIP_ will be -1 if an end-of-file occurred, a TENEX error number if an error occurred, and zero otherwise.

### SINI

"STRING" ← SINI (CHAN, MAXLENGTH, BRCHAR)

A string of characters terminated by BRCHAR or by reaching MAXLENGTH characters, whichever happens first, is read from CHAN. SINI sets _SKIP_ to -1 if the string was terminated for count; otherwise _SKIP_ will be set to BRCHAR. To determine end-of-file, examine the EOF variable for the channel (see SETINPUT).

### SIZEF

SIZE ← SIZEF (CHAN)

The size in pages of the file open on CHAN is returned. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

### UNDELETE

UNDELETE (CHAN)

The file open on CHAN is undeleted. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

## 11.4 Terminal Handling

The simplest way to write strings on the terminal is with PRINT. See page 53. The simplest way to read strings from the terminal is with INTTY. See page 79. The following detailed discussion about terminal handling will normally be of interest only to advanced programmers. The rest of this section is new.

### THE TERMINAL AS A DEVICE

We first discuss some of the problems in using the terminal as a device (i.e., when device "TTY:" is opened by OPENFILE or a similar function). Since Sail has various functions for reading strings, reals, and integers from an arbitrary device, this can be a useful feature.

TENEX provides quite general teletype service. However, the lack of a default system line editor creates some problems. Note the proliferation of line editors in the many commonly used TENEX programs. Some of them, such as the INTERLISP editor, are carefully and cleanly written. Most TENEX utility programs, however, work quite poorly and inconsistently with regard to the controlling terminal.

The TOPS-10 system has a simple line editor.

On a standard Teletype device, the standard TOPS-10 editor activates on a carriage return, altmode, control-G, or control-Z. ASCII DEL ('177) deletes the previous character; control-U deletes the current line; control-R retypes the current line; and control-Z signifies end-of-file when the terminal is INITted as a device. (The SUAI display line editor also has character insertion, deletion, searching, kill-to-character, and settable activation characters.) The great virtue of this is that programs can be written in a device-independent manner. When the terminal is accessed as a device the system handles line editing.

Many TOPS-10 programs take advantage of this device-independence, using the INPUT, REALIN and INTIN functions to access the system line editor. TENEX has had no system line editor; while IMSSS and SUMEX have had a line editor in their TENEX for some time, it is not in general use.

Therefore, the features of a "system" line editor have been put into the TENEX Sail runtime system. Several schemes have been implemented in TENEX Sail as of this writing. When a channel is opened to the controlling terminal, three kinds of line editing are available: 1) a TOPS-10 style line editor, 2) a TENEX-style line editor, and 3) no line editor at all. The TOPS-10 style editor is the default with a channel opened via OPEN; the TENEX-style editor is the default when a TENEX function (such as OPENFILE or GTJFN) is used to obtain the channel. The function SETEDIT can be used to change which convention is used. More detailed description of these three kinds of editing follows.

TOPS-10 Style Editor. The OPEN function to the controlling terminal, usually "TTY" in the second argument, gets the following editing conventions for functions INPUT, INTIN and REALIN:

'25 (control-U) deletes the entire line and echoes control-G (BEL) CR LF to the terminal.

'32 (control-Z) means end-of-file, after all previous input is read in.

'33 (ESC, altmode) activates and is sent to the program as '33. This is consistent with current TOPS-10 practice. Over the years there have been several altmodes: '33, '175, and '176. On terminals that TENEX believes to be a model 33 teletype, the characters '175 and '176 are transliterated to '33 by TENEX before the Sail runtime system sees them.

'37 (US, TENEX EOL), which is found in the input buffer when CR is typed at the terminal, is transliterated to a '15 '12 (CRLF) sequence.

'177 (DEL, rubout) deletes the last character; consecutive deleted characters are echoed, surrounded by backslashes "\". (At IMSSS and SUMEX the deleted characters are removed from the screen with the DELCH JSYS, which is not supported by BBN.)

The editor activates on line feed, altmode, control-G, and control-Z.

All this means that programs written for the TOPS-10 system, accessing the controlling terminal with INPUT et al, should work with regard to teletype input. The above is also a description of the operation of INCHWL, except that control-Z is simply a break character to INCHWL.

TENEX-Style Editor. The OPENFILE, GTJFN, and GTJFNL functions to the controlling terminal set the TENEX Sail line editor to the following conventions:

IMSSS and SUMEX. These sites use the PSTIN JSYS for line editing in TENEX, with the following conventions:

'12 (linefeed) allows input to continue on the next line.

'22 (control-R) retypes the current line.

'27 (control-W) deletes a "word" (up to the next space). This prints as "←←←" on the terminal.

'30 (control-X) deletes the entire line.

'32 (control-Z) signifies end of file.

'37 (TENEX EOL) is transliterated to a '15 '12 sequence.

'177 (rubout) or '1 (control-A) deletes the last character, using the DELCH JSYS to remove it from the display (if any).

The PSTIN JSYS transliterates '175 and '176 to '33.

The editor activates on the characters defined by the PSTIN JSYS (q.v.); these include linefeed ('12 after EOL), escape ('33), control-G, control-Z.

Sites other than IMSSS and SUMEX have the following editing conventions when the channel is opened with the TENEX routines OPENFILE, GTJFN, etc.:

'22 (control-R) retypes the current contents of the buffer.

'30 (control-X) deletes the entire line and echoes CR LF to the terminal.

'32 (control-Z) signifies end-of-file.

'37 (TENEX EOL) is transliterated to a '15 '12 sequence.

'177 (rubout) or '1 (control-A) deletes the last character. Consecutive deleted characters are echoed surrounded by backslashes.

The editor activates on line feed ('12), escape ('33), control-G (7) and control-Z ('32).

This is also the action of the INTTY routine, except that control-Z is simply a break character to INTTY.

The third mode is the BBN standard mode. In this mode all characters are simply passed through. In particular, control-Z does not signify end of file, typing a rubout gives a '177, ESC gives a '33, CR gives a '37, etc. No editing is done by the system. This is the mode in which a terminal other than the controlling terminal is accessed using any of the functions.

## SETEDIT

"OLD_MODE" ← SETEDIT (CHAN, "NEW_MODE")

If CHAN is not the controlling terminal then SETEDIT is a no-op. Otherwise, it sets the line editing mode to NEW_MODE" and returns OLD_MODE, both according to the following code:

| MODE | Meaning |
|---|---|
| "D" | TOPS-10 mode, as above |
| "T" | TENEX mode, as above |
| "B" | (BBN bag)Byte(ing) mode, no editing |

Notes:

(1) MODE SETTINGS. SETEDIT does not change or access the parameters set by such functions as SFMOD, SFCOC, STPAR, TTYUP, etc. Changes made with these latter functions will affect editing.

(2) NON-CONTROLLiNG TERMINALS. Terminals other than the controlling terminal will have byte mode -- no editing.

(3) INCHWL no longer transliterates '33 to '175. Previous versions of TENEX Sail transliterated '33 to '175.

TERMINAL MODE FUNCTIONS

The routines in this section really refer to terminals only in the "mini-system" version of TENEX. The argument CHAN may be either a Sail channel number associated with a terminal, or a terminal specifier (such as '100 or '101 for the controlling terminal).

## GTTYP, STTYP

TERMINAL_TYPE ← GTTYP (CHAN, @BUFFERS);
STTYP (CHAN, AC2)

The indicated JSYS is performed. In GTTYP the additional values returned from accumulator 2 are stored into reference parameter BUFFERS.

## RFCOC, SFCOC

RFCOC (CHAN, @AC2, @AC3);
SFCOC (CHAN, AC2, AC3)

The indicated JSYS is performed.

### RFMOD, SFMOD

MODE_WORD ← RFMOD (CHAN);
SFMOD (CHAN, AC2)

A file's mode word is queried or altered using the JSYS. WARNING: some features, such as upper case conversion, that are advertised by BBN as being accomplished with the SFMOD JSYS are actually accomplished with the STPAR JSYS.

### STPAR

STPAR (CHAN, AC2)

Does the STPAR JSYS, setting to AC2.

### STI

STI (CHAN, CHAR)

Does the STI jsys (Simulate Terminal Input) to channel CHAN (usually the controlling terminal), inserting byte CHAR into the input stream.

**DATA TRANSFER**
The usual Sail routines for teletype I/O (see page 43) are available. In addition, PBIN, PBOUT, and PSOUT have been added, although they execute exactly the same code as INCHRW, OUTCHR, and OUTSTR respectively.

### INTTY

"STRING" ← INTTY

INTTY does a TENEX-style input. (Note that INCHWL does a TOPS-10 style input.) Up to 200 characters are transfered. The activation character is not appended to the string, but is put into _SKIP_. The value -1 is placed into _SKIP_ if the input is terminated for exceeding the 200 character limit.

The normal activation characters are EOL, ESC, control-Z, and control-G; however, see the section regarding line editing in TENEX Sail. At IMSSS and SUMEX this routine uses the PSTIN JSYS with the standard system break characters; no timing is available.

### PBTIN

CHAR ← PBTIN (SECONDS)

[IMSSS only.] Executes the PBTIN JSYS with timing of SECONDS.

**SUPPRESSING OUTPUT**
This new section is for advanced Sail users only, and supposes a knowledge of the pseudo-interrupt system; see the JSYS manual and the interrupt section of this manual.

The TOPS-10 system allows the user to type a control-O and suspend program output to the terminal until either another control-O is typed or program input is requested. (See [MonCom] for a complete description.) TENEX does not have this at the system level, but pseudo-interrupts provide an alternative with which the program can receive control and abort processing as well as flush output.

TENEX Sail has complete access to the TENEX pseudo-interrupt system. In order to facilitate handling of control-O an EXTERNAL INTEGER CTLOSW has been added to the TENEX Sail runtime system. If CTLOSW is TRUE then any output to the controlling terminal (device "TTY") is flushed by the following functions:

```
PBOUT
PSOUT
OUT  to a channel open to "TTY", or to '101
OUTCHR
OUTSTR
```

CTLOSW is likewise made FALSE when input is requested by any of the following:

| | | | |
|---|---|---|---|
| INCHRS | INPUT | INTIN | TTYIN |
| INCHRW | INSTR | INTTY | TTYINS |
| INCHSL | INSTRL | PBTIN | TTYINL |
| INCHWL | INSTRS | REALIN | TTYUP |

Note: functions SINI, CHARIN and CHAROUT are not affected. CTLOSW may be accessed by declaring it as an EXTERNAL INTEGER.

Here is an example of a control-O handler.

```
ENTRY; BEGIN
REQUIRE "<><>" DELIMITERS;
DEFINE  !=<COMMENT>;
! This program sets up a control-O interrupt
  using PSI channel 0, level 0.
;

EXTERNAL INTEGER CTLOSW,.SIACS;

SIMPLE PROCEDURE CTLO; BEGIN
INTEGER USERPC,PSL1,USERINST,AC1,SAVEADDR;
LABEL LEAVE;
DEFINE  PSOUT_JSYS=<'104000000076>,
        SOUT_JSYS=<'104000000053>;

  SIMPLE INTEGER PROCEDURE DEV (INTEGER JFN);
  START_CODE
    HRRZ    2,JFN;        ! THE JFN;
    SETZ    4,;
    HRROI   1,4;          ! PUT STRING IN 4;
    MOVSI   3,'200000;    ! ONLY THE DEVICE;
    JFNS;                 ! GET THE STRING;
    MOVEM   4,1;          ! CVASC("DEV");
  END;

! this is Sail immediate interrupt level.
  No dynamic strings are accessed.;

IF CTLOSW THEN
    BEGIN
        CTLOSW ← FALSE;           ! TOGGLE IT;
        RETURN;                   ! AND RETURN;
    END;
START_CODE
  MOVEI   1,'101;
  CFOBF;
END;
OUTSTR("TO
");
CTLOSW ← TRUE;              ! NO MORE OUTPUT;

! get user PC and address into LEVTAB;
START_CODE
  MOVEI   1,'400000;
  RIR;
  HLRZ    2,2;              ! LEVTAB ADDRESS;
  MOVE    2,(2);            ! PC FOR LEVEL 1;
  MOVEM   2,PSL1;
  MOVE    2,(2);            ! USER PC;
  MOVEM   2,USERPC;
END;

! return if user mode;
IF (USERPC LAND '010000000000) THEN RETURN;

! in monitor. Return if not in the middle
  of a PSOUT or (SOUT to '101);
IF NOT (
    (USERINST ← MEMORY[USERPC-1])=PSOUT_JSYS
    OR (USERINST=SOUT_JSYS AND
       ((AC1 ← MEMORY[LOCATION(PSIACS) + 1])
         = '101 OR DEV(AC1)=CVASC("TTY"))))
THEN RETURN;

! modify return so that output stops;
SAVEADDR ← (MEMORY[PSL1] LAND '777777000000)
   + LOCATION(LEAVE);
MEMORY[PSL1] SWAP SAVEADDR;
RETURN;          ! to Sail interrupt handler;

START_CODE LEAVE:  JRST @SAVEADDR; END;
END;

INTERNAL PROCEDURE INITIALIZE;
BEGIN
PSIMAP(0,CTLO,0,1);
ENABLE(0);
ATI(0,"O"-'100);
END;

REQUIRE INITIALIZE INITIALIZATION;

END;
```

## 11.5 Utility TENEX System Calls

An effort has been made to provide calls that read and write strings which may be inconvenient to perform from START_CODE. Note that the TENEX Sail compiler has the TENEX JSYS mnemonics defined in START_CODE. In START_CODE these definitions take precedence over the function calls of the same name.

### CALL

RESULT ← CALL (AC_ARG, "FUNCTION")

A limited set of CALLs is simulated by TENEX Sail. Those available are

```
EXIT
DATE
DATSAV  [IMSSS only.]
GETINF  [IMSSS only.]
GETPPN
LOGOUT
MSTIME
PJOB
PUTINF  [IMSSS only.]
RANDOM  [IMSSS only.]
RUN
RUNTIM
TIMER
```

If any other FUNCTION is specified then a continuable error message is given.

## CNDIR

CNDIR (DIRNO, "PASSWORD")

Does the CNDIR jsys, connecting to DIRNO with password "PASSWORD". If "PASSWORD" is null then the user must have connect privileges. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred.

## DIRST, STDIR

"DIRECTORY" ← DIRST (DIRNO);
DIRNO ← STDIR ("DIRECTORY", DORECOGNITION)

These routines convert between TENEX directory numbers and strings. TENEX error codes are returned in _SKIP_, which is zero if no errors occurred. For STDIR the error codes in _SKIP_ are

```
1       string does not match
2       string is ambiguous.
```

Note that DIRECTORY must be in uppercase for the STDIR JSYS.

## GJINF

JOBNO ← GJINF (@LOGDIR, @CONDIR, @TTYNO)

The job number is returned as the value of the call. Reference values are: the number of the logged directory (LOGDIR), the connected directory (CONDIR), and the TENEX Teletype number (TTYNO).

## GTAD

DT ← GTAD

The current date and time (in TENEX representation) is returned.

## IDTIM, ODTIM

DT ← IDTIM ("DATIME");
"DATIME" ← ODTIM (DT, FORMAT)

These routines convert between TENEX internal representation DT and string representation DATIME. If DT is -1 in ODTIM then the current date and time is used. If FORMAT is -1 then the format used is "TUESDAY, APRIL 16, 1974 16:33:32". For IDTIM, TENEX error codes are returned in _SKIP_, which is zero if no errors occurred. WARNING: the IDTIM JSYS is nearly an inverse to the ODTIM JSYS; however, the format returned by ODTIM with FORMAT=-1 will NOT be recognized by IDTIM unless the day ("TUESDAY, ") is first removed. Blame BBN.

## PMAP

PMAP (AC1, AC2, AC3)

Does the PMAP JSYS, using the accumulators for the arguments.

## RDSEG

RDSEG (@SEGPAGES, @BUFPAGES)

This function returns the pages which are specially used by the Sail runtime system. The starting and ending pages of the runtime segment are returned in the left and right halves, respectively, of SEGPAGES. The first and last pages used for bufferring are returned in the left and right halves of BUFPAGES. This function is escape from Sail.

Memory map, in general:

| pages | contents |
|---|---|
| (Compile time) | |
| 0-n | impure data |
| 400-450 | compiler code |
| 600-604 | START_CODE table, if needed |
| 640-670 | runtime system |
| 770-m | UDDT |
| (Run time) | |
| 0-n | impure data |
| 400-m | code and pure data |
| 600-637 | I/O buffers |
| 640-677 | runtime system |
| 770-p | UDDT |

## RUNPRG

RUNPRG ("PROGRAM", INCREMENT, NEWFORK)

This does two entirely different things depending on the value of NEWFORK. If NEWFORK is true then a new fork is created, capabilities are transmitted, and PROGRAM is run in the new fork (with the current fork suspended by a WFORK). INCREMENT is added to the entry vector location. If NEWFORK is false then the current fork is replaced with PROGRAM. In this case RUNPRG is like the TOPS-10 RUN UUO; if the INCREMENT is 1 then the program is started at the CCL address. If RUNPRG returns at all then there was a problem with the file. Remember to say .SAV as the PROGRAM extension.

## RUNTIM

RUNNING ← RUNTM (FORK, @CONSOLE)

The running time in milliseconds for FORK is returned and the console connect time is returned in CONSOLE.

# SECTION 12

# LEAP DATA TYPES

## 12.1 Introduction

In addition to the standard algol-like statements and expressions, Sail contains an associative data store and auxiliary facilities called LEAP. Sail's version of LEAP is based on the associative components of the LEAP language implemented by J. Feldman and P. Rovner as described in [Feldman].

An associative store allows the retrieval of data based on the partial specification of that data. LEAP stores associative data in the form of ASSOCIATIONS, which are ordered three-tuples of ITEMS. Associations are frequently called TRIPLES. Associations are placed in the associative store by MAKE statements and removed from the store by ERASE statements. The associative searches allow us to specify items and their position in the triple and then have the LEAP interpreter search for triples in the associative store which have the same items in the same positions. The interpreter will extract the items from such triples, which correspond to the positions left unspecified in the original search request. For example say we had triples representing the binary relation Father_of, and we had "made" associations of the form

Father_of ⊗ John ≡ Tom
Father_of ⊗ Tom ≡ Harry,
Father_of ⊗ Jerry ≡ Tom,

where Father_of, John, Tom, Harry, and Jerry are names of items. We could then perform searches to find the sons of Tom by specifying to the leap search routines that we wanted to find triples whose first component was Father_of and whose third component was Tom. Associative searches inherently produce multiple values (i.e., both Jerry and John are sons of Tom). To deal with multiple values, Leap has SETs and LISTs of items.

Items are constants. They may be created by declaration or by the function NEW. Items may have a single algebraic variable, set, list or array associated with them which is accessible by use of the DATUM construct. Declared items have names which may be used to identify them in expressions, etc. The simple variable whose value is an item is called an ITEMVAR.

## 12.2 Syntax

The following syntax is meant to REPLACE not supplement the syntax of algebraic declarations, except where noted.

```
<declaration>
      ::= <type_declaration>
      ::= <array_declaration>
      ::= <preload_specification>
      ::= <label_declaration>
      ::= <procedure_declaration>
      ::= <synonym_declaration>
      ::= <require_specification>
      ::= <context_declaration>
|     ::= <record_class_declaration>
      ::= <protect_acs declaration>
      ::= <cleanup_declaration>
      ::= <type_qualifier> <declaration>
|     ::= <sprout_default_declaration>


<simple_type>
      ::= BOOLEAN
      ::= INTEGER
      ::= LIST
      ::= REAL
|     ::= RECORD_POINTER ( <classid_list> )
      ::= SET
      ::= STRING


<itemvar_type>
      ::= ITEMVAR
      ::= <simple_type> ITEMVAR
      ::= <array_type> ARRAY ITEMVAR
      ::= CHECKED <itemvar_type>
|     ::= GLOBAL <itemvar_type>


<item_type>
      ::= ITEM
      ::= <simple_type> ITEM


<array_type>
      ::= <simple_type>
      ::= <itemvar_type>
      ::= <item_type>
```

```
<type_declaration>
     ::= <simple_type> <identifier_list>
     ::= <itemvar_type> <identifier_list>
     ::= <item_type> <identifier_list>
     ::= <array_type> ARRAY <array_list>
     ::= <array_type> ARRAY ITEM <array_list>
     ::= <type_qualifier> <type_declaration>


<array_list> -- as on page 3


<procedure_declaration>
     ::= PROCEDURE <identifier>
          <procedure_head>
          <procedure_body>
     ::= <procedure_type> PROCEDURE
          <identifier>
          <procedure_head> <procedure_body>
     ::= <type_qualifier>
          <procedure_declaration>


<procedure_type>
     ::= <simple_type>
     ::= <itemvar_type>
     ::= MATCHING <procedure_type>
     ::= MESSAGE <procedure_type>


<procedure_head> and <procedure_body> -- as
          on page 4 except:


<simple_formal_type>
     ::= <simple_type>
     ::= <itemvar_type>
     ::= ? <itemvar_type>
     ::= <simple_type> ARRAY
     ::= <itemvar_type> ARRAY
     ::= <simple_type> PROCEDURE
     ::= <itemvar_type> PROCEDURE


<preload_specification>, <synonym_declaration>,
          <label_declaration>,

and <require_specification> as on page 3


<context_declaration> as on page 101
```

## 12.3 Semantics

ITEM GENESIS

Although items are constants, they must be created before they can be used. Items may be created in three ways:

1) A Declared Item may created by declaration of an identifier to be of type ITEM.

2) An item may be created with the NEW construct (see page 98).

3) A bracketed triple item is created by the MAKEing of a bracketed triple (see MAKE, page 90).

Items of type 1 and 2 are the same except those of type 1 may be referred to by the identifier that is associated with them. For example one may say ... ITEM DAD; ... X←DAD; .... NOTE: DAD is the name of an item, not a variable! Saying DAD←X is just as illegal as saying 15←X.

Items of type 3 are different from those of type 1 and 2. Discussion of them will be left until the creation of associations with the MAKE statement is discussed (page 90).

SCOPE OF ITEMS

Items do not obey the traditional Algol scope rules. All declared items are allocated in the outer block. All other items are allocated dynamically. All items exist until a DELETE (<item expression>) is done on them (see page 90 for the details of DELETE), or until the outer block is exited at the end of the program. HOWEVER, the identifiers of declared items (type 1 above) DO obey scope rules. After exiting the block in which item X was declared, it will be impossible to refer to X by its declared name. However, X may have been stored in an itemvar, associations, etc. and thus still be retrieved and used.

Warning: items in recursive procedures behave differently from variables in recursive procedures. At each recursive call of a procedure, the local variables are reinstantiated (unless they were declared OWN). Items are constants. There is never more than one instantiation of an item around at a time.

## DATUMS OF ITEMS

An item of type 1 or 2 may have an associated variable, called its DATUM. The Datum of an item is like any variable; it may be declared to have any type that a variable may have, except the type itemvar. Because an item may have only one datum from its creation until its death, we frequently will say the "type of an item" referring to the type of the datum. RESTRICTIONS: It is currently impossible to make either items or their datums either Internal or External. However, the effect of External items can be duplicated by manipulating the order in which items are declared (see page 87). OWN is not applicable as items are constants, not variables. Items of type ARRAY must be declared with constant bounds since they are allocated upon entering the outer block.

Example declarations of items with datums:

```
INTEGER ITEM FATHER_OF;
STRING ITEM FOO;
INTEGER ARRAY ITEM NAMES [1:4, 1:8]; COMMENT note
      the specification of the array's dimensions;
SHORT REAL ITEM POINT;

EXTERNAL ITEM BLAT;        COMMENT illegal;
ITEMVAR ITEM BLAT;         COMMENT illegal;
STRING ITEMVAR ITEM BLAT;  COMMENT illegal;
REAL PROCEDURE ITEM BLAT;  COMMENT illegal;
PROCEDURE ITEM BLAT;       COMMENT illegal,
                           use ASSIGN;
```

The syntax for variable includes the Datum construct. That is, if AGE is a declared an Integer Item, then DATUM(AGE) behaves exactly like an Integer variable. If ARR is declared as

```
STRING ARRAY ITEM ARR [2:4, 1:9*2]
```

then DATUM(ARR) is a string array with two dimensions of the declared size. A new array may not be assigned to the Datum of ARR, though of course the individual elements of the array may be changed. Datums obey the same type checking and type conversion rules that the algebraic variables of Sail do. For example, when a string is assigned to an integer datum, the integer stored in the integer datum is the ASCII of the first character of the string.

## ITEMVARS

An Itemvar is a variable whose value is an Item. Just as the statements "X←3; Y←X" and "Y←3" are equivalent with respect to Y, the statements "X←DAD; Y←X" and "Y←DAD" are equivalent with respect to Y, if X and Y are itemvars, DAD an item. The distinction between itemvars and items is identical to the distinction between integer variables and integers. An integer variable may only contain an integer and a variable declared ITEMVAR may only contain an item. This may be confusing since historically, integer variables have always been called INTEGER rather than INTEGERVAR.

Properly speaking, one should have INTEGERVAR ARRAYs instead of INTEGER ARRAYs. Originally, Sail only allowed ITEMVAR ARRAYs. However, so many people found this confusing that now one may say ITEM ARRAY, and it will be interpreted to mean ITEMVAR ARRAY. Similarly, an Item procedure is exactly the same as an itemvar procedure.

An itemvar may contain items of any type. However, when one says DATUM (ITMVR) where ITMVR is an itemvar, the compiler must know the type of the datum of the item (i.e. the type of the item) contained in the itemvar so that the the correct conversions, etc. may be done. Thus, one may declare itemvars to have the same types that are legal for items. If one has declared STRING ITEMVAR ITMVR, then the compiler assumes that you have stored an string item in ITMVR, and and will treat DATUM (ITMVR) as a string variable.

An Itemvar may be declared CHECKED if the user desires the type of itemvar checked against the type of the datum of the item expressions assigned to it. That is, only a string item could be stored in a Checked String Itemvar. If the itemvar is not declared Checked, it may have an item of any type assigned to it and their types need not match at all. This can be very dangerous. For example, an integer array item might be assigned to a string itemvar. When the datum of this itemvar is later assigned to an integer variable, say, Sail will try to treat the array header as a string pointer and get very confused. The runtime routine TYPEIT, page 123, returns a code for the type of its argument, and can be useful for avoiding type matching errors with un-checked itemvars.

GLOBAL itemvars are a special kind for SUAI global model users. Global model operation allows several jobs to share a data segment, and GLOBAL itemvars are used to build the data structures in this segment. MESSAGE procedures are also related to global model operations. These features have fallen into disuse.

EXTERNAL, OWN and INTERNAL Itemvars are legal. SAFE applies to either the array of an array itemvar, the array of an itemvar array, or both arrays of an array itemvar array.

Itemvars obey traditional Algol block structure. Upon exiting the block of their declaration, their names are unavailable and their storage is reallocated. However, the item stored in an itemvar is not affected -- it continues to exist until DELETEd or until the end of the program.

Itemvars are initialized to the special item ANY at the beginning of one's program.

SETS AND LISTS

Sets and Lists are collections of items. There are two distinctions between Sets and Lists: a list may contain multiple occurrences of any item while a set contains at most a single instance of an item. Second, the order in which items appear within a list is completely within the control of the user program, while with a set, the order is fixed by the internal representation of the items. Lists and Sets do not care what type if any the datums of their members are.

List and Set Arrays, Itemvars, Items, and Procedures are all legal, as well as External, Own and Internal Sets and Lists. Like itemvars, the scope of Set and List variables is the block they were declared in. Exiting that block does not destroy the items stored in the departed sets or lists.

ASSOCIATIONS

Perhaps the most important form of storage of items is the Association, or TRIPLE. Triples of items may be written into or retrieved from a special store, the associative store. The method of storage of these triples is designed to facilitate fast and flexible retrieval. Sail uses approximately two words of storage for each triple in the associative store. There is at most one copy of a triple in the store at any time. Once a triple has been stored in the associative store, its component items can not be changed, although an approximation to this can be obtained by erasing the association then making a new association with the altered components. You will note there is no syntax for declaring a triple. Triples can only be created with the MAKE statement. In the examples which follow, a triple is represented by :

$$A \otimes O \equiv V$$

where A, O, and V represent the items stored in the association. The associative store is accessed by the FOREACH statement, derived sets, and binding triples (see Searching the Associative Store, page 91).

PROCEDURES

Itemvar, Item, List, and Set procedures all exist. Itemvar procedures may be CHECKED if one desires the item RETURNed to have the same type as the type of the Itemvar procedure. Otherwise, the compiler only checks to see that the value returned to an itemvar procedure is an item.

Every type except Item may be used in formal parameter declarations; items are constants yet parameters always have something assigned to them in the procedure call. Since you can't assign something to a constant, you can't have item parameters.

WARNING: when using Checked Reference Itemvar formals, no type checking is performed as the actual is assigned to the formal at the procedure call. However, type checking will only be done during the procedure, and when the formal is assigned to the actual upon the (normal) exit of the procedure.

IMPLEMENTATION

Each Item is represented by a unique integer in the compiler. The numbers are assigned in the order the items are declared, e.g. the first declared item gets 1, the second gets 2, etc. (Actually, Sail has already declared 8 items that it needs, so user item numbers start with 9. REQUIRE n ITEM_START changes the number at which user items start (only useful for SUAI global model users). Lexical nesting is not observed; it is only the sequence in which the declarations are scanned that determines their numbers. The NEW function does not affect this assignment of numbers. Items created by the New function are assigned the next available number at the time of the execution of the New.

Those who use separately compiled procedures (see page 12) may wish to have declared items common to both programs. However, Internal and External items do not exist. The same effect may be achieved by carefully declaring the desired items in the same order in both programs so that their numbers match. The message "Warning -- two programs with items in them." will be issued at the begining of execution, and may be ignored if you are certain the items are declared in the same relative positions. No checking of names, types, arrays bounds, etc. is done, so be very careful.

Items occupy no space (neither does the constant integer 15). The numbers ascribed to items are stored in Itemvars and Associations. Itemvars are simply a word of storage. An association is two words of storage, one with three 12 bit bytes, each containing the number of one of the items of the association, and a second word containing two pointers relating the association to the associative search structure. Since the number of an item must fit in 12 bits, the number of items is limited to about 4090.

The number of an item may be retrieved from the item as a integer with the predeclared function CVN (<item_expression>). The item represented by a certain integer may be retrieved by the predeclared function CVI (<algebraic_expression>). CVN and CVI should only be used by those who know what they're doing and have kept themselves up to date on changes in Leap.

# SECTION 13

# LEAP STATEMENTS

## 13.1 Syntax

```
<leap_statement>
     ::= <leap_assignment_statement>
     ::= <leap_swap_statement>
     ::= <set_statement>
     ::= <list_statement>
     ::= <associative_statement>
     ::= <foreach_statement>
     ::= <suc_fail_statement>


<leap_assignment_statement>
     ::= <itemvar_variable> ←
           <item_expression>
     ::= <set_variable> ← <set_expression>
     ::= <list_variable> ← <list_expression>


<leap_swap_statement>
     ::= <itemvar_variable> ↔
           <itemvar_variable>
     ::= <set_variable> ↔ <set_variable>
     ::= <list_variable> ↔ <list_variable>


<set_statement>
     ::= PUT <item_expression> IN
           <set_variable>
     ::= REMOVE <item_expression> FROM
           <set_variable>


<list_statement>
     ::= PUT <item_expression> IN
           <list_variable>
           <location_specification>
     ::= REMOVE <item_expression> FROM
           <list_variable>
     ::= REMOVE ALL <item_expression> FROM
           <list_variable>


<location_specification>
     ::= BEFORE <element_location>
     ::= AFTER <element_location>


<element_location>
     ::= <item_expression>
     ::= <algebraic_expression>


<associative_statement>
     ::= DELETE ( <item_expression> )
     ::= MAKE <triple>
     ::= ERASE <triple>


<triple>
     ::= <item_expression> ⊗ <item_expression>
           ⊗ <item_expression>


<foreach_statement>
     ::= FOREACH <binding_list> SUCH THAT
           <element_list> DO <statement>
     ::= NEEDNEXT <foreach_statement>


<binding_list>
     ::= <itemvar_variable>
     ::= <binding_list> , <itemvar_variable>


<element_list>
     ::= <element>
     ::= <element_list> AND <element>


<element>
     ::= <item_expression> IN
           <list_expression>
     ::= ( <boolean_expression> )
     ::= <retrieval_triple>
     ::= <matching_procedure_call>


<retrieval_triple>
     ::= <ret_trip_element> ⊗
           <ret_trip_element>
           ⊗ <ret_trip_element>


<ret_trip_element>
     ::= <item_expression>
     ::= <derived_set>


<matching_procedure_call>
     ::= <procedure_call>
```

<suc_fail_statement>
::= SUCCEED
::= FAIL

## 13.2 Restrictions

SUCCEED and FAIL statements must be lexically nested inside a matching procedure to be legal.

## 13.3 Semantics

ASSIGNMENT STATEMENTS
Assignment statements in Leap are similar to those in Algol. Itemvars, Set variables, and List variables may be assigned item, set and list expressions, respectively. Only one automatic coercion is done: a set expression may be assigned to a list variable. NOTE: lists may not be assigned to set variables (use CVSET).

The type of an itemvar is checked against the type of the item expression assigned to it if and only if the itemvar is declared Checked. If a typed item is assigned to an un-Checked itemvar of different or no type, the datum is not affected. Assign an integer item to a string itemvar and the string itemvar will now contain an item with an integer datum. Sail will not know that you have in effect switched the type of the datum and will get very confused if you later try to use the datum of the itemvar; it will treat the integer as a pointer to a two word string descriptor in this case.

DATUM (X) is legal only when X is a typed item expression, namely an item expression that the compiler can discover the type of (not COP (<set>) for example). See page 128 for the BNF of typed item expressions. DATUM (X) is syntactically a variable. It has the type of the typed item expression, X. If X has an array type, then DATUM (X) should be followed by [<subscript_list>]. Appropriate coercions will be done (i.e., string to integer, integer to real, etc.) just as with regular variables in expressions. NOTE: the user is responsible for seeing that the datum of an item expression really is the type that Datum thinks it is (i.e., Datum of a Real Itemvar that has had a string item stored in it will give you garbage).

PROPS (X), where X is an item expression, is legal regardless of the type of X. X may even evaluate to a bracketed triple item, procedure item, or event item. PROPS (X) is syntactically an integer variable. It is limited to integers n where $0 \le n \le 4095$. If negative (i.e. two's complement) integers or integers larger than 4095 are assigned to a PROPS, only the right 12 bits are stored. The rest of the integer is lost.

PUT
Sets and lists are initially empty. One may put items in them with the PUT statement. "PUT <item expression> IN <set variable>" does exactly what it says.

"PUT <item expression> IN <list variable> BEFORE <algebraic expression>" evaluates the item expression, evaluates the algebraic expression and coerces it into an integer, say n, then puts the item into the list at the nth position, bumping the old nth item to the n+1th position, and so on down the list. This increases the length of the list by one. "PUT item IN list AFTER n" places the item in the n+1th position and bumps the old n+1th item down to the n+2th position, and so on. If $n < 0$ or $n > (1 + \text{length-of-list})$, then an error message is given. The special token "0" may be used in the expression for n to stand for the length of the list.

"PUT <item expression 1> IN <list variable> BEFORE <item expression 2>" cause a search to be made of the list for the item of <item expression 2>. If it is found, the item of <item expression 1> is placed in the list immediately ahead of the item found by the search. "PUT item IN list AFTER item" proceeds the same way, but puts the first item in the list immediately following the second item. If the second item is not an element of the list, a BEFORE will put the first item at the begining of the list, while an AFTER will put it at the end of the list.

REMOVE
To remove an item from a set or list, one may use REMOVE. "REMOVE item FROM set" does just what it says. If the item to be removed from the set does not occur in the set, this statement is a no-op.

"REMOVE n FROM list" removes the nth item from the list. The old n+1th item becomes the nth, and so forth. An error is indicated if $n \le 0$

or n > length-of-list. As before, ∞ should stand for the length of the list. However,

"REMOVE item FROM list" removes the first occurrence of the item from the list. If the item is not found, this statement is a no-op.

"REMOVE ALL item FROM list" removes all occurrences of the item from the list.

### DELETE

Items are represented by unique integer numbers in Sail. Due to the overwhelming desire to store an association in one word of storage, these unique numbers are limited to 12 bits. Thus the total number of items is limited to 4090. The DELETE statement allows one to free numbers for reuse. It is also the only way to get rid of an item short of exiting the program. WARNING: The Delete statement in no way alters the instances of the Deleted items which are present in sets, lists, associations, or itemvars. The user should be sure that there are no instances of the Deleted item occurring in itemvars, sets, lists or associations. Even saying DELETE (ITMVR) where ITMVR is an itemvar with an item to be deleted in it will not remove the item from ITMVR; one must be careful to change the contents of ITMVR before using it again.

### MAKE

The MAKE statement is the only way to create Associations (Triples) and add them to the associative store. If the association already exists in the store, no alterations are made. The argument to the Make statement is a triple of item expressions:

```
MAKE item1 ⊗ item2 ≡ item3
MAKE item1 ⊗ itemvar1 ≡ NEW
MAKE itemvar_array[23] ⊗ item1 ≡ itemvar2
```

The component item expressions are evaluated left to right. The three items that the three expressions evaluate to are then formed into an association, and the association is hashed into the associative store. The item expressions must be constructive, that is, one may use the NEW function but not the ANY or BINDIT items (see NEW, page 98, ANY, page 99, and BINDIT, page 99).

### BRACKETED TRIPLE ITEMS

Items may be created by declaration, by the NEW function, or by using BRACKETED TRIPLEs in Make statements. A Bracketed Triple item may not have a datum, but may have a PROPS or a PNAME (see page 124 for pnames, page 89 for props). Instead, a Bracketed Triple item has an Association connected to it. One creates a Bracketed Triple item by executing a Make statement:

```
MAKE item1 ⊗ [item2⊗item3≡item4] ≡ item5
```

where the itemN are item expressions. "[item2⊗item3≡item4]" is the Bracketed Triple item, and of course need not always be the second component of the association. The association connected to the Bracketed Triple item is "item2 ⊗ item3 ≡ item4". The above Make statement actually creates two triples and one item. Namely, the associations

```
item1 ⊗ itemXX ≡ item5
item2 ⊗ item3 ≡ item4
```

and the item "itemXX" which is a Bracketed Triple item and has the second association connected to it. One can access a Bracket Triple item, with the an associative search called the Bracketed Triple Item Retrieval:

```
itmvar ← [itm2 ⊗ itm3 ≡ itm4];
    COMMENT itmvar now contains itmXX;
```

The Bracket Triple construct may be used in any expression. See page 92.

Having "itmXX", one may access the items of the association connected to with the predeclared functions FIRST, SECOND, and THIRD (see page 125 for more information on these runtime functions):

```
FIRST (itemXX) is item2
SECOND (itemXX) is item3
THIRD (itemXX) is item4
```

### ERASE

The way to remove an association from the associative store and destroy it is to ERASE it:

```
ERASE item1 ⊗ item2 ≡ item3
```

where the itemN are item expressions. The item expressions must be retrieval item

expressions; that is, one may use the ANY item but not the NEW function or the BINDIT item (see ANY, page 99, and NEW, page 98, and BINDIT page 99). Using ANY as one, two, or three of the item expressions allows many associations to be erased in one statement. If the association to be erased does not exist, Erase is a no-op.

Whenever one Erases an association, none of the items of the association are deleted. In particular, when one Erases an association that has a Bracketed Triple item as one of its components, the Bracketed Triple item is not deleted. Furthermore, the association connected to the Bracketed Triple item is not automatically erased by erasing an association containing a Bracketed Triple item. The following Erase erases only one association:

```
ERASE item1 ⊗ [item2⊗item3≡item4] ≡ item5
```

However, erasing the association connected to a Bracketed Triple deletes the item. Deleting the Bracketed Triple item DOES NOT erase the association connected to it.

## 13.4 Searching the Associative Store

Flexible searching and retrieval are the main motivations for using an associative store. It follows that this is the most important section of the Leap part of this manual. It is a rare Leap program that does not use at least one of the searches described below.

Four methods of searching the associative store exist in Sail:

Binding Booleans
Derived Sets
Bracketed Triple item retrieval
Foreach Statements

The first three are properly part of the discussion of Leap Expressions in the next chapter, but are included here for completeness.

Throughout this section we will use the following notation for an association:

A ⊗ O ≡ V

where A, O and V stand for the "attribute", "object" and "value" items of an association.

The terms "bound" and "unbound" will find heavy use in this section. Bound describes an itemvar that has an item assigned to it. Unbound describes an itemvar that, at this time in the execution of the program, has no item bound to it. The object of searching the associative store is usually to bind unbound itemvars to specific, but unknown, items. If the itemvar to be bound was declared Checked, then type checking will be done, and the appropriate error message will be issue if the binding item does not have the same type as the itemvar.

Throughout this section, references to item expressions will always mean retrieval item expressions. Do not use NEW in such expressions.

A hashing algorithm is used in storing and retrieving associations in Leap. The user can increase the speed of associative searching or decrease his core image by using the REQUIRE n BUCKETS construct to control the size of his associative search hash table to reflect the number of associations he will be using. A hash table will be allocated with (2↑m) hash codes where m is the smallest integer such that (2↑m) ≥ n. Sail initializes the hash size to '1000.

### BINDING BOOLEANS

A Binding Boolean searches the associative store for a specified triple, returning true if one can be found, and false otherwise. A Binding Boolean is a triple:

```
itm1 ⊗ itm2 ≡ itm3
```

where "itmN" is one of three things: an item expression, or the reserved word "BIND" followed by an itemvar, or the token "?" followed by an itemvar. An item expression as a component of the Binding Boolean means that component of the triple that the boolean finds must be the item specified by the item expression (unless the item expression evaluates to the item ANY, which specifies that any item is okay). If a "BIND" itemvar is the A, O or V of the triple, then the Binding Boolean will attempt to find an association which meets the constraints imposed by the item expression A, O or V components, and then binds to the

AD-A045 102 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 9/2
SAIL, (U)
AUG 76 J F REISER
UNCLASSIFIED STAN-CS-76-574 MDA903-76-C-0206
NL

2 OF 2
AD
A045 102

END
DATE FILMED
11-77
DDC

"BIND" itemvar the items occuring in the corresponding positions of the association that the Binding Boolean found. If no such association can be found, then the Binding Boolean returns FALSE and leaves the "BIND" itemvars with their previous values. If "?" precedes an itemvar, then the itemvar will behave like a "BIND" itemvar if it is currently contains BINDIT, but will behave like an item expression if it is bound to some other item than BINDIT. Example:

```
IF Father ⊗ ?Son ≡ ANY THEN PUT Son IN Sonset;
IF ¬Father ⊗ BIND Son ≡ Bob THEN CHILDLESS (Bob);
ERCHEK ← Father ⊗ COP(Sonset) ≡ ANY;
```

## DERIVED SETS

Derived Sets are quite simple: "Foo ⊗ Garp" where Foo and Garp are item expressions, is the set of all items X such that Foo ⊗ Garp ≡ X exists. "Garp ≡ Sister" is the set of all items X such that X ⊗ Garp ≡ Sister exists. "Foo ' Sister" is the set of all items X such that Foo ⊗ X ≡ Sister exists. Examples:

```
Dadset ← Father ⊗ ANY;
Danson ← Father ' Dan;
News ← (Son ≡ Dad) ∩ attset;
```

ANY specifies "I don't care" to the search. BINDIT has no special meaning to the search, and behaves like any other items. Since BINDIT can never appear in an association, this means the set returned will always be the empty set PHI.

## BRACKETED TRIPLE ITEM RETRIEVAL

A Bracketed Triple item can be referenced by specifying the association it is connected to. For example,

```
Itmvar ← [itm1 ⊗ itm2 ≡ ANY]
PUT [ANY ⊗ ANY ≡ ANY] IN Bracset
IF Foo ⊗ Garp ≡ [itm1 ⊗ itm2 ≡ ANY] THEN ...
Itmvar ← [itm1 ⊗ [itm2 ⊗ itm3 ≡ itm4] ≡ itm5]
```

where itmN is any item expression not containing NEW or BINDIT. ANY means you don't care what item occupies that component. If the designated Bracketed Triple is not found then BINDIT is returned and no error message is given.

## THE FOREACH STATEMENT

This statement is the heart of Leap. It is similar to the FOR statement of Algol in that a statement is executed once for each binding of a variable. In this semi-schematic example,

```
FOREACH X SUCH THAT <element> AND ... AND
   <element> DO <statement>;
```

the <statement> is executed once for each binding of the itemvar X. The <element>s in the element list (i.e. <element> AND...AND <element>) determine the bindings of the itemvar, and hence how many times the <statement> is executed. If the <element>s are such that there is no binding possible for X, then the <statement> is never executed. Like a Sail FOR statement, one may use DONE, NEXT, and CONTINUE within the <statement>. As before, when one uses a NEXT inside the loop, the word NEEDNEXT must precede the FOREACH of the Foreach that one wants checked and possibly terminated. See pages 18, 19, and 19 for more information about Done, Next, and Continue.

Restriction: Jumping (i.e. with a GO TO) into a Foreach is illegal. However, it is legal to jump out of a Foreach, or to jump around within the same Foreach.

Foreach statements differ from For statements in that more than one itemvar may be included to be given bindings:

```
FOREACH X, Y, Z SUCH THAT <element>....
```

X, Y, and Z are called Foreach itemvars. Just as one must declare the integer I before using it in the Sail For statement

```
FOR I ← 1 STEP 2 UNTIL 21 DO...
```

so must one declare Foreach itemvars before using them in Foreaches. Foreach itemvars are no more than normal itemvars receiving special assignments; they may have any type. If a Foreach itemvar that has been declared Checked is assigned an item by the search that has a different type than the Checked itemvar, an error message will result.

Foreach itemvars differ from For variables in a more radical way. It is possible to specify to

the Foreach that a certain Foreach itemvar be a variable to the search only on the condition that that the itemvar contains the special item BINDIT at the time the Foreach is called. One precedes such itemvars with the "?" token. For example:

```
FOREACH ?X, ? Y, Z SUCH THAT <element>....
```

If X contains BINDIT but Y does not when this Foreach starts execution, then the search will be conducted exactly as if the statement

```
FOREACH X,Z SUCH THAT <element>....
```

were the Foreach specified. The itemvar X will then act just like an ordinary, non-foreach itemvar that was bound previous to the Foreach. All Foreach itemvars may be "?" itemvars if this is desired.

There are four different types of <element> that may be used in foreach element lists:

Set Membership
Boolean Expressions
Retrieval Triples
Matching Procedures

The order of the <element>s in the element list is very important, as we shall see.

Terminology: we say that a certain binding of the the Foreach itemvars "satisfies" an <element>. If that binding satisfies each <element> of the element list, then we say it "satisfies the associative context". A fancy way of refering to the element list is "associative context". We also refer to the collection of bindings that satisfy the associative context as the "satisfier group" of the Foreach.

The execution of a Foreach proceeds as follows. After initialization, the Foreach proceeds with a search specified by the first <element> of the element list. If a binding can be found that satisfies the first <element>, the Foreach proceeds forward to the new <element> of the list and tries to satisfy it, and so on. When the Foreach can not satisfy an <element>, it "backs up" to the previous element and tries to get a different binding. If it can't find satisfaction there, it backs up again and tries again to get a different binding. When a Foreach proceeds forward off the end of the element list (i.e. the associative context is satisfied) then the <statement> is executed, and the Foreach backs up to the last <element> of the element list. When the Foreach backs up off the left end of the element list, the Foreach is exited.

When a Foreach is exited by backing up off the left, the Foreach itemvars are restored to the last satisfier group bound to them, regardless of what the <statement> may have done. If the associative context was never satisfied, then the Foreach itemvars have the values that they had before the Foreach. When a Foreach is exited with a GO TO, DONE, or RETURN, the Foreach leave the itemvars with the bindings they had at the GO TO, or whatever, including any modifications that the <statement> may have made to them.

THE LIST MEMBERSHIP <ELEMENT>
[In the following, one may also read "set" for "list"; Sail automatically coerces set expressions into list expressions.] This <element> does not search the associative store to bind an itemvar, but merely binds it with an item of a specified list. In the Foreach,

```
FOREACH X | X IN L DO <statement>;
```

(here we have used the Sail synonym "|" for "SUCH THAT"), the Foreach itemvar X is bound successively to each element of the set L, starting at the beginning of the list. If an item occurs n times in L, then X will be bound to that item n times in the course of the Foreach. Thus, the number of satisfiers to the above Foreach is LENGTH (L).

In the current implementation of Leap, there is a difficulty that should be pointed out. If inside the <statement>, one changes L by list assignment, Removes, etc. in such a way as to remove the next item of the list that the Foreach itemvar would have been bound to, Leap may go crazy. Foreach searches look one ahead and save a pointer to the next items to be bound to the Foreach itemvars. This allows one to remove the items of the current bindings of the Foreach itemvars from lists or whatever, but makes other removals hazardous. For example,

```
FOREACH X | X IN L DO REMOVE X FROM L;
```

will work, but

```
PUT V IN L BEFORE FOO;
FOREACH X | X IN L DO  REMOVE V FROM L;
```

will probably fail. No error checking is done.

Whenever the Foreach itemvar of a list <element> has been bound previously, the list element behaves like a boolean. It does not rebind the itemvar but only checks to see that it is in the list. For example,

```
FOREACH X | X IN L AND X IN LL DO <statement>;
```

X is bound by the <element> "X IN L". <element> "X IN LL" is satisfied if the item contained in the itemvar X is in the list LL.

If two different Foreach itemvars are used with two different lists, i.e.

```
FOREACH X,Y | X IN L AND Y IN LL
DO <statement>;
```

then after execution of the <statement>, the Foreach will go back the last <element> that searches for bindings, in this case "Y IN LL" and gets a new binding for Y. It is only on failure of this search that the Foreach goes back to the first <element>, "X IN S", and gets a new binding for X. Thus the <statement> will be executed once for each possible X,Y pair. In the Foreach,

```
FOREACH X,Y | X IN L AND Y IN L ...;
```

X and Y will be bound to all possible pairs of elements in L. This includes pairs with duplicate elements, like (a,a). Different orderings of the same elements will NOT be ignored. Thus, pairs like (a,b) and (b,a) will each be a satisfier group sometime during the Foreach. Furthermore, if the list L contains duplications of the same item, identical pairs will occur in proportion to the number of duplications. That is, regardless of the duplications within the list, the number of satisfier groups to the Foreach above is LENGTH (L)↑2.

### THE BOOLEAN EXPRESSION <ELEMENT>

Any Sail boolean expression may be used as an <element> in the Associative Context of a Foreach if it is inclosed by parentheses. A Boolean Expression <element> is satisfied if it is TRUE. Note that the boolean expression must have parentheses around it.

WARNING: Foreach itemvars can not be bound by a Boolean Expression <element>. Therefore, all itemvars used in a Boolean Expression <element> must be bound by previous <element>s in the element list. A Boolean Expression <element> with unbound Foreach itemvars in it causes an error message.

### THE RETRIEVAL TRIPLE <ELEMENT>

To search the associative store with a Foreach, one uses the Retrieval Triple <element>. A Retrieval Triple is satisfied if a binding of the Foreach itemvars can be found such that the triple is an extant association. If all of the itemvars of the Retrieval Triple <element> were bound previous to the execution of the Retrieval Triple <element>, then the Triple does no further binding; it is satisfied if the specified triple is in the associative store. For example,

```
FOREACH X | FATHER ⊗ TOM ≡ X AND
   X IN PTA_SET DO <statement>;

FOREACH X | X IN PTA_SET AND
   FATHER ⊗ TOM ≡ X DO <statement>;
```

The two Foreaches have the same effect. However, in the first case, X is bound by a search of the associative store for any triple that has FATHER as its attribute component, and TOM as its object component. When such a triple is found, X is bound to the item that is the value component. Then, if X is in the PTA_SET, the Foreach lets the statement execute. If X is not in PTA_SET, then the Foreach backs up and tries to find another triple with FATHER as its attribute and TOM as its value. In the second Foreach, X is bound with an item from PTA_SET, then the associative store is checked to see that the triple FATHER⊗TOM≡x, where x is the binding of X, is in the store. If it is, the <statement> is executed, otherwise the Foreach backs up and gets a different item from PTA_SET and binds that to X. Assuming that Tom has only one father, the first search is much faster.

Using ANY in a Retrieval Triple indicated that you don't care what item occupies that position. For instance, in

```
FOREACH X | FATHER ⊗ ANY ≡ X DO <statement>;
```

X is bound successively to all fathers. However, if the associative store included the following three associations,

```
FATHER ⊗ KAREN ≡ PAUL
FATHER ⊗ LYNN ≡ PAUL
FATHER ⊗ TERRY ≡ PAUL
```

then X would be bound to PAUL only once, not thrice. BINDIT has no special meaning to the search. Since BINDIT can never appear in an association, a Retrieval Triple containing it will cause the search to always fail.

Different kinds of associative searches proceed with different efficiencies. Listed below in order of decreasing efficiency are the various forms of Retrieval Triple <element>s that are legal. A, O, and V represent either bound Foreach itemvars or items from explicit item expressions in the triple. x, y, and z represent unbound Foreach itemvars or the item ANY. (note that x ⊗ x ≡ V is really x ⊗ O ≡ V, and so on). The two forms of the List Membership <element> are included for comparison.

| | |
|---|---|
| x IN L | All items x in the list L. |
| A ⊗ O ≡ x | Only the value is free. |
| x ⊗ y ≡ V | Attribute and object are free. |
| A IN L | Verification that item A is in list L. |
| A ⊗ O ≡ V | Verification that the triple is in the store. |
| A ⊗ x ≡ V | Only the object is free. |
| x ⊗ O ≡ V | Only the attribute is free. |
| A ⊗ x ≡ y | Object and value are free. |
| x ⊗ O ≡ y | Attribute and value are free. |
| x ⊗ y ≡ z | Attribute, value and object are free. |

Note that MAKEing an association inside a Foreach may or may not affect subsequent bindings. For example, in

```
FOREACH X,Y | Link ⊗ X ≡ Y DO
      MAKE Link ⊗ X ≡ Newlink;
```

it is uncertain whether Y will ever receive Newlink as its binding or not.

The A, O, and V used in a Retrieval Triple of a Foreach may be a derived set expressions as well as item expressions. For example,

```
FOREACH X, Y | Link ⊗ (Father⊗Y) ≡ X DO ...;
```

ERASE in the <statement> of a Foreach that binds any of its itemvars with Retrieval Triples may cause problems. This is similar to REMOVE used in Foreaches with List Membership <element>s controling some bindings. ERASE can only be guaranteed to to work safely if the association erased is the one we just got a binding from, e.g.

```
FOREACH X | A ⊗ O ≡ X DO ERASE A ⊗ O ≡ X;
```

or if the association erased could not possible be used for a binding of a Foreach itemvar, such as,

```
FOREACH X | Link ⊗ X ≡ Node DO
      ERASE Node ⊗ X ≡ ANY;
```

Foreaches look one ahead to the next binding of its itemvars, and leaves a pointer to those associations. If you Erase any of those associations, the Foreach gets lost in the boondocks. No error checking is done.

However, as long as the associative store is not changed during the execution of the Foreach, a Retrieval Triple will not itself repeat a particular set of bindings that it bound before.

### THE MATCHING PROCEDURE <ELEMENT>

Matching Procedures are the most general search mechanism in Leap. They also provide a convenient method of writing coroutines.

A MATCHING Procedure is very similar to a boolean procedure (in fact outside of Foreach associative contexts, it behaves like a boolean procedure and may be called within expressions, etc.). It must be declared type MATCHING. It may not be declared SIMPLE. The formal parameters of a Matching Procedure may include zero or more "?" itemvars (pronounced "question itemvars") which may have any datum type but may not be VALUE or REFERENCE. These parameters correspond roughly to either call by value or call by reference, depending on the actual parameter when the procedure is called. When the actual parameter is an item expression or a bound itemvar the parameter is equivalent to a value parameter. However, if the actual parameter is an unbound Foreach itemvar, then the parameter is treated as a reference parameter, and on entry is is initialized to the special item BINDIT.

Matching Procedures are exited by SUCCEED and FAIL statements instead of RETURN statements. When used outside of an associative context, SUCCEED corresponds to RETURN(TRUE) and FAIL corresponds to RETURN(FALSE) [this is not strictly true when

the matching procedure is sprouted as a process -- see page 106]. Inside an associative context, Succeed and Fail determine whether the Foreach is to proceed to the next <element> of the element list or to backup to the previous <element> of the element list. When the Foreach backs up into a Matching Procedure, the procedure is not recalled, but resumed at the statement following the last Succeed executed. On the other hand, when a Foreach proceeds forward into a Matching Procedure, the procedure is called, not resumed.

When a Matching Procedure is the last <element> of the associative context, Succeeding will cause the <statement> to be executed; the Foreach then backs up into the Matching Procedure, and the Matching Procedure is resumed at the statement following the Succeed. When a Matching Procedure is the first <element> of an associative context, Failing will exit the Foreach.

WARNING: Matching procedures are implemented as processes and two calls of the same matching procedure may share the same memory unless the procedure is declared RECURSIVE. See Memory Accessible to a Process, page 105.

If a Matching Procedure is explicitly SPROUTed as a process then the Matching Procedure can be made running by a RESUME. In such a case the item sent by RESUME is returned as the value of the SUCCEED or FAIL statement which suspended the Matching Procedure, just as though SUCCEED or FAIL were an item procedure. (In fact Succeed and Fail always return an item value, but the value is ANY except in this special case.) Being Resumed is the only was in which a Matching Procedure can be reactivated after a FAIL.

When a Matching Procedure is used exterior to the associative context of a Foreach, one may use "BIND" in the call preceding those actuals which one wishes bound regardless of their current binding. Preceding the actual with "?" will have the save effect as "BIND" if the current value of the itemvar is BINDIT, and will have no effect otherwise (the procedure will not attempt to find it a binding).

That is all there is to Matching Procedures. Their power lies in the using them cleverly.

The following program illustrates techniques one may use with matching procedures by simulating the List Membership and Retrieval Triple <element>s with matching procedures.

```
RECURSIVE MATCHING PROCEDURE INLIST
        (? ITEMVAR X; LIST L);
BEGIN "INLIST"
   COMMENT THIS PROCEDURE SIMULATES THE CONSTRUCT
      X ∈ L FOR ALL CASES EXCEPT THE SIMPLE
      PREDICATE BINDIT∈L;
   IF X ≠ BINDIT THEN
      BEGIN WHILE LENGTH (L) DO IF X = LOP (L)
         THEN BEGIN SUCCEED; DONE END;
         FAIL
      END;
   WHILE LENGTH (L) DO BEGIN X←LOP (L);
         SUCCEED END;
END "INLIST";

MATCHING PROCEDURE TRIPLE (? ITEMVAR A, O, V);
BEGIN "TRIPLE"
   DEFINE BINDING (A)="(A=BINDIT)";
   SET SET1; INTEGER INDX;
   RECURSIVE PROCEDURE SUCC_SET (REFERENCE
                  ITEMVAR X; SET S1);
      WHILE LENGTH (S1) DO BEGIN X← OP (S1);
         SUCCEED END;

   INDX ← 0;
   IF BINDING (A) THEN INDX ← 1;
   IF BINDING (O) THEN INDX ← INDX + 2;
   IF BINDING (V) THEN INDX ← INDX + 4;
   CASE INDX OF
   BEGIN [0] "A&O≡V" IF A&O≡V THEN SUCCEED;
      [1] "?&O≡V" SUCC_SET (A, O≡V);
      [2] "A&?≡V" SUCC_SET (O, A'V);
      [3] "?&?≡V" BEGIN SET1 ← ANY ≡ V;
         WHILE (LENGTH (SET1)) DO
            BEGIN A ← LOP (SET1);
               SUCC_SET (O, A'V) END END;
      [4] "A&O≡?" SUCC_SET (V, A&V);
      [5] "?&O≡?" BEGIN SET1 ← O ≡ ANY;
         WHILE (LENGTH (SET1)) DO
            BEGIN A ← LOP (SET1);
               SUCC_SET (V, A&O) END END;
      [6] "A&?≡?" BEGIN SET1 ← A ' ANY;
         WHILE (LENGTH (SET1)) DO
            BEGIN O ← LOP (SET1);
               SUCC_SET (V, A&O) END END;
      [7] "?&?≡?"
         USERERR(0, 1, "ANY&ANY≡ANY IS IN BAD TASTE")
   END;
END "TRIPLE";
```

# SECTION 14

# LEAP EXPRESSIONS

## 14.1 Syntax

```
<leap_expression>
     ::= <item_expression>
     ::= <set_expression>
     ::= <list_expression>

<item_expression>
     ::= <item_primary>
     ::= [ <item_primary> ⊗ <item_primary> ≡
          <item_primary> ]

<item_primary>
     ::= NEW
     ::= NEW ( <algebraic_expression> )
     ::= NEW ( <set_expression> )
     ::= NEW ( <list_expression> )
     ::= NEW ( <array_name> )
     ::= ANY
     ::= BINDIT
     ::= <item_identifier>
     ::= <itemvar_variable>
     ::= <list_expression> [
          <algebraic_expression> ]
     ::= <itemvar_procedure_call>
     ::= <resume_construct>
     ::= <interrogate_construct>

<itemvar_procedure_call>
     ::= <procedure_call>

<list_expression>
     ::= <list_primary>
     ::= <list_expression> & <list_expression>

<list_primary>
     ::= NIL
     ::= <list_variable>
     ::= {{ <item_expr_list> }}
     ::= ( <list_expression> )
     ::= <list_primary> [ <substring_spec> ]
     ::= <set_primary>

<item_expr_list>
     ::= <item_expression>
     ::= <item_expr_list> , <item_expression>

<set_expression>
     ::= <set_term>
     ::= <set_expression> ∪ <set_term>

<set_term>
     ::= <set_factor>
     ::= <set_term> ∩ <set_factor>

<set_factor>
     ::= <set_primary>
     ::= <set_factor> - <set_primary>

<set_primary>
     ::= PHI
     ::= <set_variable>
     ::= {item_expr_list}
     ::= ( <set_expression> )
     ::= <derived_set>

<derived_set>
     ::= <item_expression>
          <associative_operator>
          <item_expression>

<associative_operator>
     ::= ⊗
     ::= '
     ::= ≡

<itemvar_variable>
     ::= <variable>

<set_variable>
     ::= <variable>

<list_variable>
     ::= <variable>

<leap_relational>
     ::= <item_expression> IN
          <set_expression>
     ::= <item_expression> IN
          <list_expression>
```

```
    ::= <item_expression>
        <item_relational_operator>
        <item_expression>
    ::= <set_expression>
        <set_relational_operator>
        <set_expression>
    ::= <list_expression>
        <list_relational_operator>
        <list_expression>
    ::= <triple>


<item_relational_operator>
    ::= =
    ::= ≠


<set_relational_operator>
    ::= =
    ::= ≠
    ::= <
    ::= >
    ::= ≤
    ::= ≥


<list_relational_operator>
    ::= =
    ::= ≠
```

## 14.2 Semantics

ITEM EXPRESSIONS
Itemvars and itemvar arrays may be used in item expressions just as algebraic variables and algebraic arrays are used in algebraic expressions. Itemvars and itemvar arrays are initialized to the special Sail item ANY.

Items may be retrieved from sets and lists with the Sail functions COP and LOP. COP (<set expression or list expression>) yields the item which is the first element of the set or list that the set or list expression evaluated to. LOP also yields the first item of the set or list, but removes that item from the set or list. Because LOP changes the contents of the set or list that is its argument, it can only accept set or list variables, not expressions. See page 48.

List element designators may be used as itemvars in expressions. For example, if RECORD is a list, and ITMVR an itemvar,

```
RECORD[5] ← ITMVR;
ITMVR ← RECORD[∞-1];
RECORD[∞] ← RECORD[1];
```

are all legal. The special token "∞" means the length of the list when used in this context. The contents of the square brackets may be any algebraic expression as long as it evaluates to an integer n where $1 \le n \le$ LENGTH (list).

<list_expression> [<algebraic_expression>]
returns a particular element of a list, but may not appear on the left of an assignment expression, because assignment must be to variables.

NEW
The function NEW creates an item at execution time. Since space must be allocated at loading for various tables, one must indicate approximately how may NEW items he will create (the compiler counts the declared items for you). Therefore, one should say "REQUIRE n NEW_ITEMS" where n is some integer less than 4090 (the maximum number of items allowed in Sail). n may be larger than the actual number of New items created, but the excess will be wasted space. If $0 < n < 50$, you get tables for 50 New items anyway.

NEW may take an argument. In this case, the datum of the created item is preloaded with the value passed as argument. If this argument is algebraic, set or list, then the datum will be of the same type. No type conversions are done when passing the algebraic argument. NEW will also accept an array name as argument. In this case, the created item will be of the type array. In fact, the array cited as argument will be copied into the newly created array. The new array will have the same bounds and number of dimensions as the array cited as argument. This array will not disappear even if the block that the original array was declared in is exited. It will only be deallocated if the item is deleted.

NEW in an item expression makes that item expression a "constructive item expression". Constructive item expressions are illegal in some places, namely anywhere that attempts to gets an item from an existing structure (i.e., ERASE, REMOVE, and Associative searches). It is usually clear whether or not a constructive item expression is illegal.

### ANY

Some associative searches may need only partial specification. The ANY item is used to specify exactly which parts of the specification are "don't cares"'s. Examples:

```
FOREACH X SUCH THAT Father ⊗ X ≡ ANY DO ...
IF Father ⊗ BIND X ≡ ANY THEN ...
```

ANY in an item expression makes that item expression a "retrieval item expression". This is the opposite of a constructive item expression, and is illegal anywhere the statement is creating new structure, namely, a MAKE statement. Thus, ANY is legal everywhere items are, except a MAKE statement.

### BINDIT

Like ANY, BINDIT specifies no constraints on the associative search. However, BINDIT has a special meaning to some searches, namely the Binding Boolean and Matching Procedures (depending on how they're written). An itemvar containing BINDIT will be bound by the search to an item of the association that the search found. For example:

```
X ← BINDIT;
IF Father ⊗ ? X ≡ Bob
   THEN PUT X IN Bobfatherset;
```

Like ANY, BINDIT is illegal in MAKE statements. In certain associative searches, namely the ERASE statement, the Bracketed Triple Item retrieval expression, and the Retrieval Triple <element> of a Foreach, inclusion of BINDIT will cause the search to always fail, because BINDIT can appear in no association.

### TYPES AGAIN

The compiler can determine the type of items when the item expression is a typed itemvar, a typed itemvar procedure, a declared item with a type, a typed itemvar array, or a NEW with an argument. When the compiler can determine the type of the item expression, then and only then is it legal to use the Datum construct on the item expression or to assign the item expression to a Checked itemvar. For example, the following are ILLEGAL:

```
DATUM (COP (<set>))
DATUM (RECORD[∞]);   COMMENT RECORD is a list;
CHEC ← NEW;  COMMENT CHEC is a Checked itemvar;
```

### SET AND LIST EXPRESSIONS

Three rather standard operations are implemented for use with sets. These are union (∪), intersection (∩), and subtraction (-). These operators have the standard mathematical interpretations. The only possible confusion pertains to subtractions: if we perform the set operation

```
set1 - set2
```

and if there is an instance of an item x in set2 but not in set1, the subtraction proceeds and no error message is given.

If one considers a list to be a string of items, then concatenation and taking sublists suggest themselves as likely list operations. The syntax and semantics for sublisting and list concatenation are identical with those of strings, with the natural exception that the results are lists, and not strings. There is also a difference in that if the indices to the substringer do not make sense, an error message is generated rather than setting of the _SKIP_ variable. Examples:

```
LISTVAR ← LISTVAR[2 TO ∞-1];
LISTVAR ← LISTVAR[9 FOR 2*N];
LISTVAR ← LISTVAR[1 FOR 2] & LISTVAR[3 TO ∞];
```

One may generate sets with

```
{item1, item2, item3}
```

and may generate lists with

```
{{item1, item1, item2, item3}}.
```

Sets are initialized to the empty set, PHI. Lists are initialized to the null list, NIL. Initialization occurs at the beginning of the execution of the program. Sets and list are reinitialized on entering the blocks of their declaration only when such blocks are in recursive procedures.

### DERIVED SETS

Derived sets are really sets of answers to questions which search the associative memory. The conventions are:

```
a ⊗ b      -- the set of all x such that a ⊗ b ≡ x
a ≡ b      -- the set of all x such that x ⊗ a ≡ b
a ' b      -- the set of all x such that a ⊗ x ≡ b
```

BOOLEANS
Several boolean primaries are implemented for comparing sets, lists, and items. In the following discussion, "ix" means item expression, "se" means set expressions, and "le" means list expression. These are:

1) Set and List Membership. The boolean "ix IN se" evaluates the set or list expression, and returns TRUE if the item value specified by the item expression is a member of the set or list.

2) Association Existence. The binding boolean "ix ⊛ ix ≡ ix", where the ix are item expressions or itemvars preceded by ? or BIND, returns TRUE if a binding of the BIND itemvars (and ? itemvars that contained BINDIT) can be found such that the association exists in the associative store. See page 91 for more information on binding bo[illegible].

3) [illegible]

| | |
|---|---|
| ix = ix | obvious interpretation |
| ix ≠ ix | obvious interpretation |
| se1 < se2 | true if se1 is a proper subset of se2 |
| se1 ≤ se2 | true if se1 is identical to se2 or is a proper subset of se2 |
| se1 = se2 | obvious interpretation |
| se1 ≠ se2 | obvious interpretation |
| se1 > se2 | equivalent to se2 < se1 |
| se1 ≥ se2 | equivalent to se2 ≤ se1 |
| le1 = le2 | obvious interpretation |
| le1 ≠ le2 | obvious interpretation |

PNAMES
For those desire them, each item may have a string, called its PNAME, linked with it. This is completely independent of the Datum construct. New items and Bracketed Triple items are created with NULL strings as their Pnames. One may delete an item's Pname with the DEL_PNAME function which takes an item expression as its argument. One may give a Pnameless item a Pname with the NEW_PNAME procedure, which takes an item expression and a string as its arguments. CVSI will give you the Pname of an item, and CVIS with give you the item with the specified Pname. No two items may have the same Pname. Pnames do not follow Algol scope rules. See page 124 to find out how to use the above four functions.

If you would like your declared items to have Pnames that are the same as the identifier used in their declaration, say "REQUIRE PNAMES" or "REQUIRE n PNAMES" before their declaration at the beginning of the program. The n is an estimate of the number of dynamically created items with pnames you will use -- this causes tables for n pnames to be allocated at compile time rather than runtime, thus making your program more efficient.

PROPS
Any item may have a PROPS. This is an extra 12 bits of storage (frequently used for bits). PROPS (X) where X is an item expression is exactly an integer variable in its syntax. See page 89 for further information on props.

# SECTION 15

# BACKTRACKING

## 15.1 Introduction

Backup or backtracking is the ability to "back up" execution to a previous point. Sail facilitates backtracking by allowing one to REMEMBER, FORGET, or RESTORE variables in the data type CONTEXT.

## 15.2 Syntax

<context_declaration>
    ::= CONTEXT <id_list>
    ::= CONTEXT ARRAY <array_list>
    ::= CONTEXT ITEM <id_list>
    ::= CONTEXT ITEMVAR <id_list>

<backtracking_statement>
    ::= <rem_keyword> <variable_list>
        <rem_preposition> <context_variable>

<rem_keyword>
    ::= REMEMBER
    ::= FORGET
    ::= RESTORE

<rem_preposition>
    ::= IN
    ::= FROM

<variable_list>
    ::= <vari_list>
    ::= ( <vari_list> )
    ::= ALL
    ::= <context_variable>

<vari_list>
    ::= <vari>
    ::= <vari_list> , <vari>

<vari>
    ::= <variable>
    ::= <array_identifier>

<context_variable>
    ::= <variable>

<array_identifier>
    ::= <identifier>

<context_element>
    ::= <context_variable> : <variable>

## 15.3 Semantics

THE CONTEXT DATA TYPE
A context is essentially a storage place of undefined capacity. When we REMEMBER a variable in a context, we remember the name of the variable along with its current value (if an array, values). If we remember a value which we have already remembered in the named context, we destroy the old value we had remembered and replace it with the current value of the variable. Values can be given back to variables with the RESTORE statement.

Context variables are just like any other variables with respect to scope. Also, at execution time, context variables are destroyed when the block in which they were declared is exited in order to reclaim their space. Context arrays, items, and itemvars are legal (items and itemvars are part of Leap). NEW( <context variable> ) is legal (NEW is also part of Leap).

RESTRICTIONS:

1. Context procedures do not exist. Use context itemvar procedures instead.

2. Context variables may only be passed by reference to procedures (i.e., contexts are not copied).

3. Contexts may not be declared "GLOBAL" (shared between jobs - SUAI only).

4. +, *, /, and all other arithmetic operators have no meaning when

applied to Context variables. Therefore, context variable expressions always consist only of a context variable.

The empty context is NULL_CONTEXT. Context variables are initialized to NULL_CONTEXT at program entry.

### REMEMBER

To save the current values of variables, list them, with or without surrounding parentheses, in the remember statement. All of an array will be remembered if subscripts of an array are not used, otherwise, only the value indicated will be remembered. If a variable has already been remembered in context, its value is replaced by the current value. If one wants to update all the variables so far remembered in this context, one may say

```
REMEMBER ALL IN <context>.
```

If you have several contexts active,

```
REMEMBER CNTXT1 IN CNTXT2
```

will note the variables Remembered in CNTXT1, and automatically Remember their CURRENT values in CNTXT2.

### RESTORE

To restore the values of variables that were saved in a context, list them (with or without surrounding parentheses) in a restore statement. Restoring an array without using subscripts causes as much of the array that was remembered to be restored magically to the right locations in the array. You can remember a whole array, then restore all or selected parts (e.g. RESTORE A[1, 2] FROM IX;). If you remembered only A[1, 2], then restoring A will only update A[1, 2]. RESTORE ALL IN IX will of course restore all the variables from IX. RESTORE CNTXT1 FROM CNTXT2 will act like a list of the variables in CNTXT1 was presented to the Restore instead of the identifier CNTXT1.

Astute Leap users will have noted that the syntax for variables includes Datum(typed itemvar) and similar things. If one executes REMEMBER DATUM (typed_item_expression_1) IN CNTXT, then RESTORE DATUM (<item_expression_2>) FROM CNTXT will give an error message unless the <typed_item_expression_2> returns the same item as <typed_item_expression_1>.

WARNING!!! Restoring variables that have been destroyed by block exits will give you garbage. For example, the following will blow up:

```
BEGIN "BLOWS UP"
   CONTEXT J1;
   INTEGER J;
   BEGIN INTEGER ARRAY L[1:J];
         REMEMBER J, L IN J1;
   END;
   RESTORE ALL FROM J1;
END "BLOWS UP";
```

### FORGET

The forget statement just deletes the variable from the context without touching the current variable's value. Variables remembered in a context should be forgotten before the block in which the variables were declared is exited. FORGET ALL FROM X1 and FORGET CNTXT1 FROM CNTXT2 work just as the similar Restore statements work, only the variables are Forgotten instead of Restored.

### IN_CONTEXT

The runtime boolean IN_CONTEXT returns true if the specified variable is in the specified context. For details, see page 51.

### CONTEXT ELEMENTS

Context elements provide a convenient method of accessing a variable that is being remembered in a context. Examples of context elements:

```
CNTXT_VARI : SOME_VARI
DATUM (CNTXT_ITEM) : SOME_VARI
CNTXT_AR[2,3] : ARRY[4]
DATUM (CNTXT_VARI : ITMVR)
CNTXT_VARI : DATUM(ITMVR)
```

A context element is syntactically and semantically equivalent to a variable of the same type as the variable following the colon. For the complete syntax of variables, see page 128. Assignments to context elements change the Remembered value (i.e., X←5; REMEMBER X IN C; C:X←6; RESTORE X FROM C; will leave X with the value 6).

As with the Restore statement, one may not use Context Elements of variables destroyed by block exits.

RESTRICTIONS: (1) One may not Remember Context Elements. (2) Passing Context Elements

by reference to procedures that change contexts is dangerous. Namely, if the procedure Forgets the element that was passed to it by reference, then the user is left with a dangling pointer. A more subtle variation of this disaster occurs when the Context element passed is an array element. If the procedure Remembers the array that that array element was a part of, the formal that had the array element Context Element passed to it is left with a dangling pointer.

# SECTION 16

# PROCESSES

## 16.1 Introduction

A PROCESS is a procedure call that may be run independently of the main program. Several processes may "run" concurrently. When dealing with a multi-process system, it is not quite correct to speak of "the main program". The main program is actually a process itself, the main process.

This section will deal with the creation, control, and destruction of processes, as well as define the memory accessible to a process. The following section will describe communication between processes.

## 16.2 Syntax

```
<process_statement>
     ::= <sprout_statement>

<sprout_statement>
     ::= SPROUT ( <item_expression> ,
          <procedure_call> ,
          <algebraic_expression> )
     ::= SPROUT ( <item_expression> ,
          <procedure_call> )
     ::= SPROUT ( <item_expression> ,
          <apply_construct> )

<sprout_default_declaration>
     ::= SPROUT_DEFAULTS <integer_constant>
```

## 16.3 Semantics

**STATUS OF A PROCESS**
A process can be in one of four states: terminated, suspended, ready, or running. A terminated process can never be run again. A suspended process can be run again, but it must be explicitly told to run by some process that is running. Since Sail is currently implemented on a single processor machine, one cannot really execute two procedures simultaneously. Sail uses a scheduler to swap processes from ready to running status. A running process is actually executing, while a ready process is one which may be picked by the scheduler to become the running process. The user may retrieve the status of a process with the execution time routine PSTATUS, page 109.

**SPROUTING A PROCESS**
One creates a process with the SPROUT statement:

```
SPROUT (<item>, <procedure call>, <options>)
SPROUT (<item>, <procedure call>)
```

<item> is a construction item expression (i.e. do not use ANY or BINDIT). Such an item will be called a process item. The item may be of any type; however, its current datum will be writen over by the SPROUT statement, and its type will be changed to "process item" (see TYPEIT, page 123). RESTRICTION: A user must never modify the datum of a process item.

<procedure call> is any procedure call on a regular or recursive procedure, but not a simple procedure. This procedure will be called the process procedure for the new process.

<options> is an integer that may be used to specify special options to the SPROUTer. If <options> is left out, 0 will be used. The different fields of the word are as follows:

| BITS | NAME | DESCRIPTION |
|---|---|---|
| 14-17 | QUANTUM (X) | Q ← IF X=0 THEN 4 ELSE 2↑X; The process will be given a quantum of Q clock ticks, indicating that if the user is using CLKMOD to handle clock interrupts, the process should be run for at most Q clock ticks, before calling the scheduler. (see about CLKMOD, page 120 for details on making processes "time share"). |
| 18-21 | STRINGSTACK (X) | S ← IF X=0 THEN 16 ELSE X*32; The process will |

be given S words of string stack.

22-27 PSTACK (X) P←IF X=0 THEN 32 ELSE X*32; The process will be given P words of arithmetic stack.

28-31 PRIORITY (X) P ← IF X=0 THEN 7 ELSE X; The process will be given a priority of P. 0 is the highest priority, and reserved for the Sail system. 15 is the lowest priority. Priorities determine which ready process the scheduler will next pick to make running.

32 SUSPHIM If set, suspend the newly sprouted process.

33 Not used at present.

34 SUSPME If set, suspend the process in which this sprout statement occurs.

35 RUNME If set, continue to run the process in which this sprout statement occurs.

The names are defined in the file ⊂SUAI⊃SYS:PROCES.DEF, which one may require as a source file. Options words may be assembled by simple addition, e.g. RUNME + PRIORITY (3) + PSTACK (2).

DEFAULT STATUS: If none of bits 32, 34, or 35 are set, then the process in which the sprout statement occurs will revert to ready status, and the newly sprouted process will become the running process.

The default values of QUANTUM, STRINGSTACK, PSTACK, and PRIORITY are stored in the system variables DEFQNT, DEFSSS, DEFPSS, and DEFPRI respectively. These values may be changed. The variables are declared EXTERNAL INTEGERs in ⊂SUAI⊃SYS:PROCES.DEF.

### SPROUT_DEFAULTS

If one of the "allocation" fields of the options word passed to the SPROUT routine -- i.e., QUANTUM, STRINGSTACK, PSTACK, or PRIORITY -- is zero, then SPROUT will look at the corresponding field of the specified <integer_constant> of the SPROUT_DEFAULTS for the procedure being sprouted. If the field is non-zero then that value will be used; otherwise the current "system" default will be used.

NOTE: SPROUT_DEFAULTS only applies to "allocations", i.e., the process status control bits (e.g. SUSPME) are not affected. Example:

```
RECURSIVE PROCEDURE FOO;
 BEGIN
 SPROUT_DEFAULTS STRINGSTACK (10);
 INTEGER XXX;
 :
 :
 END;
 :
SPROUT (P1, FOO, STRINGSTACK (3));
SPROUT (P2, FOO);
COMMENT P1 will have a string stack of 3*32 words.
 P2 will have a string stack of 10*32 words;
```

### MEMORY ACCESSIBLE TO A PROCESS

A process has access to the same global variables as would a "normal" call of the process procedure at the point of the SPROUT statement. For example, suppose you Sprouted a process in the first instantiation of a recursive procedure and immediately suspended it. Then in another instantiation of the procedure, you resumed the process. Since each recursive instantiation of a procedure creates and initializes new instances of its local variables, the process uses the instances of the recursive procedure's locals that were current at the time of the SPROUT, namely those of the first instantiation.

Sail will give you an error message whenever the global variables of a process are deallocated but the process still exists. Usually, this means that when the block in which the process procedure was declared is exited, the corresponding process must be terminated (one can insure this by using a small Cleanup procedure that will TERMINATE the fated process or JOIN it to the current one -- see about Cleanup, page 10, Terminate, page 107, and Join, page 109). When the process procedure has been declared inside a recursive procedure, things become a bit more complex. As mentioned above, the process takes its

globals from the context of the Sprout statement. Therefore, it is only in the instantiation of the recursive procedure that executed the Sprout that trouble can occur. For example,

```
RECURSIVE PROCEDURE TENLEVEL (INTEGER I);
BEGIN "TROUBLE"
   PROCEDURE FOO;
   ; COMMENT does nothing;

   IF I=5 THEN SPROUT (NEW, FOO, SUSPHIM);

     COMMENT sprouts FOO on the 5th
     instantiation of TENLEVEL, then
     immediately suspends it;

   IF I<10 THEN TENLEVEL (I+1);
   RETURN;

     COMMENT assuming TENLEVEL is called
     with I=0, it will do 10 instantiations,
     then come back up;

END "TROUBLE";
```

TENLEVEL will nest 10 deep, then start returning. This means "TROUBLE" will be exited five times will no ill effects. However, when Sail attempts to exit "TROUBLE" a sixth time, it will be exiting a block in which a process was sprouted and declared. It will generate the error message, "Unterminated process dependent on block exited".

The construct DEPENDENTS (<block_name>), where <block_name> is a string constant, produces a set of process items. The process items are those of all the processes which depend on the current instance of the named block -- i.e. all processes whose process procedures obtain their global variables from that block (via the position of the process procedure's declaration, or occasionaly via the location of the Sprout in a nest of recursive procedure instantiations). This construct may be used together with a CLEANUP procedure (see page 10) to avoid having a block exit before all procedures dependent on it have been terminated.

If one Sprouts the same non-recursive procedure more than once (with different process items, of course), the local variables of the procedure are not copied. In other words, "X←5" in process A will store 5 in the same location that "X←10" in process B would store 10. If such sharing of memory is undesirable, declare the process procedure RECURSIVE, and then new instances of the local variables of the procedure will be created with each Sprout involving that procedure. Then "X" in process A will refer to a different memory location than "X" in process B.

### SPROUT APPLY

The <procedure call> in a SPROUT statement may be an APPLY construct. In this case SPROUT will do the "right" thing about setting up the static link for the APPLY. That is, "up-level" references by the process will be made to the same variable instances that would be used if the APPLY did not occur in a SPROUT statement. (See page 115.)

However, there is a glitch. The sprout mechanism is not yet smart enough to find out the block of the declaration of the procedure used to define the procedure item. It would be nice if it did, since then it could warn the user when that block was exited and yet the process was still alive, and thus potentially able to refer to deallocated arrays, etc. What the sprout does instead is assume the procedure was declared in the outer block. This may be fixed eventually, but in the meantime some extra care should be taken when using apply in sprouts to avoid exiting a block with dependents. Similarly, be warned that the "DEPENDENTS (<blockid>)" construct may not give the "right" result for sprout applies.

### SPROUTING MATCHING PROCEDURES

When a matching procedure is the object of a Sprout statement, the FAIL and SUCCEED statements are interpreted differently than they would be were the matching procedure called in a Foreach or as a regular procedure. FAIL is equivalent to RESUME (CALLER (MYPROC), CVI (0)). SUCCEED is equivalent to RESUME (CALLER (MYPROC), CVI (-1)).

### SCHEDULING

One may change the status of a process between terminated, suspended and ready/running with the TERMINATE, SUSPEND, RESUME, and JOIN constructs discussed above, and the CAUSE and INTERROGATE constructs discussed in the next chapter. This section will

describe how the the status of processes may change between ready and running.

Whenever the currently running process performs some action that causes its status to change (to ready, terminated, or suspended) without specifying which process is to be run next, the Sail process scheduler will be invoked. It chooses a process from the pool of ready processes. The process it chooses will be made the next running process. The scheduling algorithm is essentially round robin within priority class. In other words, the scheduler finds the highest priority class that has at least one ready process in it. Each class has a list of processes associated with it, and the scheduler choses the first ready process on the list. This process then becomes the running process and is put on the end of the list. If no processes have ready status, the scheduler looks to see if the program is enabled for any interrupts (see Interrupts, page 117). If the program is enabled for some kind of interrupt that might still happen (not arithmetic overflow, for instance), then the scheduler puts the program in interrupt wait. After the interrupt is dismissed, the scheduler tries again to find a ready process. If no interrupts that may still happen are enabled, and there are no ready processes, the error message "No one to run" is issued.

The rescheduling operation may be explicitly invoked by calling the runtime routine URSCHD, which has no parameters.

POLLING POINTS

Polling points are located at "clean" or "safe" points in the program; points where a process may change from running to ready and back with no bad effects. Polling points cause conditional rescheduling. A polling point is an efficient version of the statement:

```
IF INTRPT ∧ ¬NOPOLL THEN
    BEGIN INTRPT←0; URSCHD END;
```

INTRPT is an external integer that is used to request rescheduling at the next polling point. It is commonly set by the deferred interrupt routine DFRINT (for all about deferred interrupts, see page 121) and by the clock interrupt routine CLKMOD (for how to make processes time share, see page 120). The user may use INTRPT for his own purposes (carefully, so as not to interfere with DFRINT or CLKMOD) by including the declaration "EXTERNAL INTEGER INTRPT", then assigning INTRPT a non-zero value any time he desires the next polling point to cause rescheduling. NOPOLL is another external integer that is provided to give the user a means of dynamically inhibiting polling points. For example, suppose one is time sharing using CLKMOD. In one of the processes, a point is reached where it becomes important that the processes not be swapped out until a certain tight loop is finished up. By assigning NOPOLL (which was declared an EXTERNAL INTEGER) a non-zero value, the polling points in the loop are efficiently ignored. Zeroing NOPOLL restores normal time sharing.

A single polling point can be inserted with the statement POLL. The construct

```
REQUIRE n POLLING_INTERVAL
```

where n is a positive integer, causes polling points to be inserted at safe points in the code, namely: at the start of every statement provided that at least n instructions have been emitted since the last polling point, after every label, and at the end of every loop. If $n \leq 0$ then no further polling points will be put out until another Require n (n>0) Polling_Interval is seen.

## 16.4 Process Runtimes

### TERMINATE

TERMINATE (PROC_ITM)

The process for which PROC_ITM is the process item is terminated. It is legal to terminate a terminated process. A terminated process is truly dead. The item may be used over for anything you want, but after you have used it for something else, you may not do a terminate on it. Termination of a process causes all blocks of the process to be exited.

## SUSPEND

ITM ← SUSPEND (PROC_ITM)

The process for which PROC_ITM is the process item is suspended. If the process being suspended is not the currently running process then the item returned is ANY. In cases such as

```
X ← SUSPEND (MYPROC);
```

where the process suspends itself, it might happen that this process is made running by a RESUME from another process. If so, then X receives the SEND_ITM that was an argument to the RESUME.

One may suspend a suspended process. Suspending a terminated process will cause an error message. If the process being suspended is the currently running process (i.e. the process suspends itself), then the scheduler will be called to find another process to run. A process may also be suspended as the result of RESUME or JOIN.

## RESUME

RET_ITM ← RESUME (PROC_ITM,
SEND_ITM, OPTIONS(0))

RESUME provides a means for one process to restore a suspended process to ready/running status while at the same time communicating an item to the awakened process. It may also specify what its own status should be. It may be used anywhere that an itemvar procedure is syntactically correct. When a process which has suspended itself by means of a RESUME is subsequently awakened by another resume, the SEND_ITM of the awakening RESUME is used as the RET_ITM of the RESUME that caused the suspension. For example, suppose that process A has suspended itself:

```
STARTINFO ← RESUME (Z, NEED_TOOL);
```

If later a process B executes the statement

```
INFOFLAG ← RESUME (A, HAMMER);
```

then B will suspend itself and A will become the running process. A's process information will be updated to remember that it was awakened by B (so than the runtime routine CALLER can work). Finally, A's RESUME will return the value HAMMER, which will be assigned to STARTINFO. If A had been suspended by SUSPEND or JOIN then the SEND_ITM of B's RESUME is ignored.

A process that has been suspended in any manner will run from the point of suspension onward when it is resumed.

OPTIONS is an integer used to change the effect of the RESUME on the current process (MYPROC) and the newly resumed process.

| BITS | NAME | DESCRIPTION |
|---|---|---|
| 33-32 | READYME | If 33-32 is 1, then the current process will not be suspended, but be made ready. |
| | KILLME | If 33-32 is 2, then the current process will be terminated. |
| | IRUN | If 33-32 is 3, then the current process will not be suspended, but be made running. The newly resumed process will be made ready. |
| 34 | | This should always be zero. |
| 35 | NOTNOW | If set, this bit makes the newly resumed process ready instead of running. If 33-32 are not 3, then this bit causes a rescheduling. |

DEFAULT: If none of bits 35 to 32 are set, then the current process will be suspended and the newly resumed process will be made running. At SUAI include a REQUIRE "SYS:PROCES.DEF" SOURCE_FILE in your program to get the above bit names defined. Options may then be specified by simple addition, e.g. KILLME + NOTNOW.

## CALLER

PROCITEM ← CALLER (PROCITEM2)

CALLER returns the process item of the process that most recently resumed the process referred to PROCITEM2. PROCITEM2 must be the process item of an unterminated process, otherwise an error message will be issued. If PROCITEM2's process has never been called, then the process item of the process that sprouted PROCITEM2 is returned.

## DDFINT

DDFINT

A polling point is SKIPE INTRPT; PUSHJ P, DDFINT. DDFINT suspends the current process (but leaves it ready to run), then calls the scheduler; DDFINT is like SUSPEND (MYPROC, IRUN+NOTNOW).

## JOIN

JOIN (SET_OF_PROCESS_ITEMS)

The current process (the one with the JOIN statement in it) is suspended until all of the processes in the set are terminated. WARNING: Be very careful; you can get into infinite wait situations.

1. Do not join to the current process; since the current process is now suspended, it will never terminate of its own accord.
2. Do not suspend any of the joined processes unless you are assured they will be resumed.
3. Do not do an interrogate-wait in any of the processes unless you are sure that the event it is waiting for will be caused (page 110).

## MYPROC

PROCITEM ← MYPROC

MYPROC returns the process item of the process that it is executed in. If it is executed not inside a process, then MAINPI (the item for the main process) is returned.

## PRISET

PRISET (PROCITM, PRIORITY)

PRISET sets the priority of the process specified by PROCITM (an item expression that must evaluate to the process item of a non-terminated process) to the priority specified by the integer expression PRIORITY. Meaningful priorities are the integer between 1, the highest priority, to 15, the lowest priority. Whenever a rescheduling is called for, the scheduler finds the highest priority class that has at least one ready process in it, and makes the first process on that list the running process. See about the scheduler, page 107.

## PSTATUS

STATUS ← PSTATUS (PROCITM)

PSTATUS returns an integer indicating the status of the process specified by the item expression PROCITM.

| | |
|---|---|
| -1 | running |
| 0 | suspended |
| 1 | ready |
| 2 | terminated |

## URSCHD

URSCHD

URSCHD is essentially the Sail Scheduler. When one calls URSCHD, the scheduler finds the highest priority class that has at least one Ready process in it. Each class has a list of processes associated with it, and the scheduler choses the first ready process on the list. This process then becomes the running process and is put on the end of the list. If no processes have ready status, the scheduler looks to see if the program is enabled for any interrupts. If the program is enabled for some kind of interrupt that may still happen (not arithmetic overflow, for instance), then the scheduler puts the program into interrupt wait. After the interrupt is dismissed, the scheduler tries again to find a ready process. If no interrupts that may still happen are enabled, and there are no ready processes, the error message "No one to run" is issued.

# SECTION 17

# EVENTS

## 17.1 Syntax

```
<event_statement>
     ::= <cause_statement>
     ::= <interrupt_statement>


<cause_statement>
     ::= CAUSE ( <item_expression> ,
          <item_expression> ,
          <algebraic_expression> )
     ::= CAUSE  ( <item_expression> ,
          <item_expression> )


<interrogate_construct>
     ::= INTERROGATE ( <item_expression> ,
          <algebraic_expression> )
     ::= INTERROGATE ( <item_expression> )
     ::= INTERROGATE ( <list_expression> ,
          <algebraic_expression> )
     ::= INTERROGATE ( <list_expression> )
```

## 17.2 Introduction

The Sail event mechanism is really a general message processing system which provides a means by which an occurrence in one process can influence the flow of control in other processes. The mechanism allows the user to classify the messages, or "event notices", into distinct types ("event types") and specify how each type is to be handled.

Any leap item may be used as an event notice. An event type is an item which has been given a special runtime data type and datum by means of the runtime routine:

MKEVTT (et)

where et is any item expression (except ANY or BINDIT). With each such event type Sail associates:

1. a "notice queue" of items which have been "caused" for this event type.
2. a "wait queue" of processes which are waiting for an event of this type.
3. procedures for manipulating the queues.

The principle actions associated with the event system are the CAUSE statement and the INTERROGATE construct. Ordinarily these statements cause standard Sail runtime routines to be invoked. However, the user may substitute his own procedures for any event type (see User Defined Cause and Interrogate procedures, page 112). The Cause and Interrogate statements are here described in terms of the Sail system supplied procedures.

## 17.3 Sail-defined Cause and Interrogate

### THE CAUSE STATEMENT

```
CAUSE (<event type>, <event notice>, <options>)
CAUSE (<event type>, <event notice> )
```

<event type> is an item expression, which must yield an event type item. <event notice> is an item expression, and can yield any legal item. <options> is an integer expression. If <options> is left out, 0 is used.

The Cause statement causes the wait queue of <event type> to be examined. If it is non-empty, then the system will give the <event notice> to the first process waiting on the queue (see about the WAIT bit in Interrogate, below). Otherwise, <event notice> will be placed at the end of the notice queue for <event type>.

The effect of Cause may be modified by the appropriate bits being set in the options word:

| BITS | NAME | DESCRIPTION |
|---|---|---|
| 35 | DONTSAVE | Never put the <event item> on the notice queue. If there is no process on the wait queue, this makes the cause statement a no-op. |
| 34 | TELLALL | Set the status of all processes waiting for this event to READY. |
| 33 | RESCHEDULE | Reschedule as soon as possible (i.e., immediately after the cause procedure has completed executed). |

DEFAULT: If bits 35 to 33 are 0, then the either a single process is awakened from the wait queue, or the event is placed on the notice queue. The process doing the Cause continues to run. At SUAI, REQUIRE "SYS:PROCES.DEF" SOURCE_FILE to get the above bit names defined. Options can then be constructed with simple addition, e.g. DONTSAVE + TELLALL.

### THE INTERROGATE CONSTRUCT - SIMPLE FORM

```
<itemvar> ← INTERROGATE (<event type>, <options>)
<itemvar> ← INTERROGATE (<event type>)
```

<event type> is an item expression, which must yield an event type item. <options> is an integer expression. If <options> is left out, 0 is used.

The notice queue of <event type> is examined. If it is non-empty, then the first element is removed and returned as the value of the Interrogate. Otherwise, the special item BINDIT is returned.

<options> modifies the effect of the interrogate statement as follows:

| BITS | NAME | DESCRIPTION |
|---|---|---|
| 35 | RETAIN | Leave the event notice on the notice queue, but still return the notice as the value of the interrogate. If the process goes into a wait state as a result of this interrogate, and is subsequently awakened by a Cause, then the DONTSAVE bit in the Cause statement will override the RETAIN bit in the Interrogate if both are on. |
| 34 | WAIT | If the notice queue is empty, then suspend the process executing the interrogate and put its process item on the wait queue. |
| 33 | RESCHEDULE | Reschedule as soon as possible (i.e., immediately after execution of the interrogate procedure). |
| 32 | SAY_WHICH | Creates the association EVENT_TYPE ⊗ <event notice> ≡ <event type> where <event type> is the type of the event returned. Useful with the set form of the Interrogate construct, below. |

DEFAULT: If bits 35 to 32 are 0, then the interrogate removes an event from the event queue, and returns it. If the event queue is empty, BINDIT is returned and no waiting is done; the process continues to run. At SUAI, use a REQUIRE "SYS:PROCES.DEF" SOURCE_FILE to get the names defined; use simple addition to form options, e.g. RETAIN + WAIT.

### THE INTERROGATE CONSTRUCT - SET FORM

```
<itemvar> ← INTERROGATE (<event type set>)
<itemvar> ← INTERROGATE (<event type set>, <options>)
```

<event type set> is a set of event type items. <options> is an integer expression. If it is left out, 0 will be used.

The set form of interrogate allows the user to examine a whole set of possible event types. This form of interrogate will first look at the notice queues, in turn, of each event type in <event type set>. If one of these notice queues is non-empty, then the first notice in that queue will be remved and that notice will be returned as the value of the Interrogate. If all the notice queues are empty, and WAITing is not specified in the options word, then BINDIT will be returned. When the WAIT bit is set, the process doing the interrogate gets put at the

end of the wait queues of each event type in <event type set>. Then, when a notice is finally available, the process is removed from all of the wait queues before returning the notice. Note that the option SAY_WHICH provides a means for determining which event type produced the returned notice.

## 17.4 User-defined Cause and Interrogate

By executing the appropriate runtime routine, the user can specify that some non-standard action is to be associated with CAUSE or INTERROGATE for a particular event type. Such user specified cause or interrogate procedures may then manipulate the event data structure directly or by themselves invoking the primitives used by the Sail Cause and Interrogate constructs. User defined Cause and Interrogate are not for novice programmers (this is an understatement).

### EVENT TYPE DATA STRUCTURE

The datum of an event type item points to a six word block of memory. This block contains the following information:

| WORD | NAME | TYPE | DESCRIPTION |
|---|---|---|---|
| 0 | NOTCQ | LIST | The list of all notices pending for this event type. |
| 1 | WAITQ | LIST | The list of all processes currently waiting for a notice of this type. |
| 2 | --- | --- | Procedure specifier for the user specified cause procedure (zero if system procedure is to be used). |
| 3 | --- | --- | Procedure specifier for the user specified interrogate procedure (zero if system procedure is to be used). |
| 4 | USER1 | INTEGER | Reserved for user. |
| 5 | USER2 | INTEGER | Reserved for user. |

The appropriate macro definitions for these names (e.g. WAITQ (et) = "MEMORY[ DATUM (et)+1, LIST ]" ) are included in the file ⊂SUAI⊃SYS:PROCES.DEF.

### USER CAUSE PROCEDURES

A procedure to be used as a Cause procedure must have three formal value parameters corresponding to the event type, event notice, and options of the Cause. Such a procedure is associated with an event type by means of the runtime SETCP:

```
SETCP (<event type>, <procedure specifier>);
```

where <event type> must yield an event type item and <procedure specifier> is either a procedure name or DATUM (<procedure item>). For example:

```
PROCEDURE CX (ITEMVAR ET, EN; INTEGER OPT);
  BEGIN
  PRINT ("Causing ", EN,
  " as an event of type ", ET);
  CAUSE1 (ET, EN, OPT);
  END;
...

SETCP (FOO, CX);
```

Now,

```
CAUSE (FOO, BAZ);
```

would cause CX (FOO, BAZ) to be called. This procedure would print out "Causing BAZ as an event of type FOO" and then call CAUSE1. The runtime CAUSE1 (ITEMVAR etype, enot; INTEGER opt) is the Sail runtime routine that does all the actual work of causing a particular notice, enot, as an instance of event type etype. It is essentially this procedure which is replaced by a user specified cause procedure.

CAUSE1 uses an important subroutine which is also available to the user. The integer runtime ANSWER (ITEMVAR ev_type, ev_not, process_item) is used to wake up a process that has suspended itself with an interrogate. If the process named by process_item is suspended, it will be set to ready status and be removed from any wait queues it may be on. ANSWER will return as its value the options bits from the interrogate that caused the process to suspend itself. If the named process was not suspended, then ANSWER returns an integer word with bit 18 (the '400000 bit in the right half = NOJOY in ⊂SUAI⊃SYS:PROCES.DEF) set to

1. The ev_type and ev_not must be included in case the SAY_WHICH bit was on in the interrogate which caused the suspension. ANSWER has no effect on the notice queue of ev_type.

Frequently one may wish to use a cause procedure to re-direct some notices to other event types. For instance:

```
PROCEDURE CXX (ITEMVAR ET, EN; INTEGER OPT);
   BEGIN ITEMVAR OTH; LABEL C;
   IF redirecttest(ET, EN) THEN
      FOREACH OTH | OTHER_CAUSE@ET-OTH DO
         C:    CAUSE1 (ET, EN, OPT)
   ELSE  CAUSE1 (ET, EN, OPT);
   END;
```

In order to avoid some interesting race conditions, the implementation will not execute the causes at C immediately. Rather, it will save ET, EN and OPT, then, when the procedure CXX is finally exited, any such deferred causes will be executed in the order in which they were requested.

## USER INTERROGATE PROCEDURES

A user specified interrogate procedure must have two value formal parameters corresponding to the two arguments to INTERROGATE and should return an item as the value. The statement

```
SETIP (<event type>, <procedure specifier>);
```

where <event type> is an event type item, and <procedure specifier> is either a procedure name or DATUM (<procedure item>), will make the specified procedure become the new interrogate procedure for <event type>. For instance:

```
ITEMVAR PROCEDURE IX (ITEMVAR ET; INTEGER OPT);
   BEGIN ITEMVAR NOTI;
   NOTI ← ASKNTC (ET, OPT);
   PRINT ("Notice ", NOTI, " returned
      from interrogation of ", ET);
   RETURN (NOTI);
   END;
...

SETIP (FOO, IX);
```

Now,

```
... ← INTERROGATE (FOO);
```

would cause NOTI to be set to the value of ASKNTC (FOO, 0). Then the message "Notice BAZ returned from interrogate of FOO" would be printed and IX would return NOTI as its value.

The runtime ASKNTC (ITEMVAR etype; INTEGER opt) is the Sail system routine for handling the interrogation of a single event type. Essentially it is the procedure being replaced by the user interrogate procedure.

In the case of multiple interrogations, Sail sets a special bit (bit 19 = '200000 in the right half = MULTIN in ⊂SUAI⊃SYS:PROCES.DEF) in the options word before doing any of the interrogates specified by the event type items in the event type set. The effect of this bit, which will also be set in the options word passed to a user interrogate procedure, is to cause ASKNTC always to return BINDIT instead of ever waiting for an event notice. Then, if ASKNTC returns BINDIT for all event types, Sail will cause the interrogating process to Wait until its request is satisfied. If multin is not set, then ASKNTC will do the WAIT if it is told to.

# SECTION 18

# PROCEDURE VARIABLES

## 18.1 Syntax

```
<assign_statement>
     ::= ASSIGN ( <item_expr> ,
          <procedure_name> )
     ::= ASSIGN ( <item_expr> ,
          DATUM ( <item_expr> ) )

<ref_item_construct>
     ::= REF_ITEM ( <expression> )
     ::= REF_ITEM ( VALUE <expression> )
     ::= REF_ITEM ( BIND <itemvar> )
     ::= REF_ITEM ( ? <itemvar> )

<apply_construct>
     ::= APPLY ( <procedure_name> )
     ::= APPLY ( <procedure_name> ,
          <arg_list_specifier> )
     ::= APPLY ( DATUM ( <item> ) )
     ::= APPLY ( DATUM ( <item> ) ,
          <arg_list_specifier> )

<arg_list_specifier>
     ::= <list_expression>
     ::= ARG_LIST ( <expr_list> )
```

## 18.2 Semantics

ASSIGN
One may give an item a procedure "datum" using the ASSIGN statement. ASSIGN accepts as its first argument an item expression (do not use ANY or BINDIT). To this is bound the procedure identified by its name or to the "datum" of another procedure item. The procedure may be any type. However, the value it returns will only be accessible if the procedure is an itemvar or item procedure. Apply assumes that whatever the procedure left in AC 1, (the register used by all non-string procedures to return a value) on exiting is an item number. Warning: a procedure is no ordinary datum. Using DATUM on a procedure item except in the above context will not work. Use APPLY instead.

REF_ITEM
Reference items are created at run time by the REF_ITEM construct and are used principally in argument lists for the APPLY construct. The datum of a reference item contains a pointer to a data object, together with type information about that object. To create a reference item one executes

```
itm ← REF_ITEM (<expression>)
```

A NEW item is created. If the expression is (a) a simple variable or an array element, then the address will be saved in the item's datum. If the expression is (b) a constant or "calculated" expression, then Sail will dynamically allocate a cell into which the value of the expression will be saved, and the address of that cell will be saved in the datum of the item. The item is then noted as having the datum type "reference" and returned as the value of the REF_ITEM construct. One can slightly modify this procedure by using one of the following variations.

```
itm ← REF_ITEM (VALUE <expression>)
```

In this case, a temp cell will always be allocated. Thus X←3; XI←REF_ITEM (VALUE X); X←4; would cause the datum of XI to point at a cell containing 3.

```
itm ← REF_ITEM (? itmvr)
itm ← REF_ITEM (BIND itmvr)
```

where itmvr must be an itemvar or an element of an itemvar array, will cause the reference item's datum to contain information that Apply can use to obtain the effect of using "? itmvr" or "BIND itmvr" as an actual parameter in a procedure call.

ARG_LIST
The ARG_LIST construct assembles a list of "temporary" reference items that will be deleted by APPLY after the applied procedure returns. Arguments to ARG_LIST may be anything legal for REFITEM. Thus

```
APPLY (proc, ARG_LIST (foo, bar, VALUE baz))
```

is roughly equivalent to

```
tmplst ← {{REF_ITEM (foo), REF_ITEM (bar),
          REF_ITEM (VALUE baz)}};
APPLY (proc, tmplst);
WHILE LENGTH (tmplst) DO DELETE (LOP (tmplst));
```

but is somewhat easier to type. Note that the reference items created by ARG_LIST are just like those created by REF_ITEM, except that they are marked so that APPLY will know to kill them.

### APPLY

APPLY uses the items in the <arg_list_specifier>, together with the environment information from the procedure item (or from the current environment, if the procedure is named explicitly) to make the appropriate procedure call. <arg_list_specifier> is an ordinary list expression, except that each element of the list must be a reference item. The elements of the list will be used as the actuals in the procedure call. There must be at least as many list elements as there are formals in the procedure. The reference items must refer to an object of the same type as the corresponding formal parameter in the procedure being called. (EXCEPTION: if the formal parameter is an untyped itemvar or untyped itemvar array, then the reference item may refer to a typed itemvar or itemvar array, respectively.) At present, type checking (but not type coercion) is done. If the formal parameter is a reference parameter, then a reference to the object pointed to by the reference item is passed. If the formal parameter is a value parameter, then the value of the object pointed to by the reference item is used. Similarly, "?" formals are handled appropriately when the reference item contains a "?" or "BIND" reference. If the procedure to be called has no parameters, the <arg_list_specifier> may be left out.

Apply may be used wherever an itemvar procedure call is permitted. The value returned will be whatever value would normally be returned by the the applied procedure, but Apply will treat it as an item number. Care should therefore be taken when using the result of Apply when the procedure being invoked is not itself an itemvar procedure, since this may cause an invalid item number to be used as a valid item (for instance, in a MAKE). Recall that when a typed procedure (or an Apply) is called at statement level, the value it returns is ignored. Here is an example of the use of APPLY.

```
BEGIN
LIST L;INTEGER XX;
INTEGER ITEMVAR YY;ITEMVAR ZZ;
REAL ARRAY AA[1:2];
PROCEDURE FOO (INTEGER X;
   ITEMVAR Y,Z; REAL ARRAY A);
      BEGIN
      Y ← NEW (X);
      Z ← NEW (A);
      A[X]←3;
      END;
XX←0;
L ← {{REF_ITEM (XX), REF_ITEM (YY),
   REF_ITEM (ZZ), REF_ITEM (AA)}};
XX ← 2; AA[1] ← AA[2] ← 1;
APPLY (FOO, L);
COMMENT Y now contains an item whose
   datum is 2, Z contains an item whose
   datum is the  array (1.0, 1.0),
   A[1]=1.0, and A[2]=3.0;
END;
```

The variables accessed by a procedure called with APPLY may not always be what you would think they were. Temporary terminology: the "environment" of a procedure is the collection of variables, arrays and procedures accessible to it. "Environment" is not meant to include the state of the associative store or the universe of items. The environment of a procedure item is the environment of the ASSIGN, and that environment will be used regardless of the position of the APPLY. Since procedure items are untouched by block exits, yet environments are, it is possible to Apply a procedure item when its environment is gone; Sail catches most of these situations and gives an error message. Consider the following example:

```
BEGIN
ITEM P; LABEL L;
RECURSIVE PROCEDURE FOO (INTEGER J);
BEGIN "FOO"
   INTEGER I;
   PROCEDURE BAZ;
     PRINT ("J=", J, " I=", I);
   IF J=1 THEN
      BEGIN
      I←2;
      ASSIGN (P, BAZ);
      FOO (-1);
      END
   ELSE APPLY (DATUM (P));
END "FOO";
FOO (1);
L: APPLY (DATUM (P)); COMMENT will cause a
      runtime error -- see discussion below;
END
```

The effect of the program is to Assign Baz to P on the first instantiation of Foo, then Apply P on the second (recursive) instantiation. However, the environment at the time of the Assign includes {I=2, J=1} but the environment at the time of the Apply includes {I=0, J=-1} instead. At the time of the Apply, Baz is executed with the environment from the time of the Assign, and will print out

```
J=1 I=2
```

The Apply at L will cause a runtime error message because the environment of the Assign has been destroyed by the exiting of Foo.

# SECTION 19

# INTERRUPTS

## 19.1 Introduction

The interrupt facilities of Sail are based on the interrupt facilities provided by the operating system under which Sail is running. For programs running at SUAI or on TENEX this results in satisfactory interrupt operation. TOPS-10 programs are at a distinct disadvantage because the operating system does not prevent interrupt handlers from being interrupted themselves. At SUAI the Sail system uses new-style interrupts [Frost]; programs may also enable for old-style interrupts and the two will work together provided that the same condition is not enabled under both kinds. On TENEX the pseudointerrupt (PSI) system is used; programs may use the interrupt system independently of Sail. Only interrupt functions pertaining to the current fork are provided. TOPS-10 interrupts are directly tied to the APRENB system; Sail and non-Sail use do not mix.

Sail gives control to the user program as soon as the operating system informs the Sail interrupt handler. This can be dangerous because the Sail runtime system may be in the middle of core allocation or garbage collection. Therefore Sail provides a special runtime DFRINT which can receive control in the restricted environment of an interrupt. DFRINT records the fact that an interrupt happened and that a particular user procedure is to be run at the next polling point (page 107), when the integrity of all runtime data structures is (normally) assured. If the Sail interrupt handler passes control to DFRINT then the user procedure (which is run at the next polling point) is called a "deferred interrupt procedure", even though the only connection it has with interrupts is the special status and priority given to it by the Sail Process machinery. If DFRINT is not used then the user procedure to which the Sail interrupt handler passes control is called an "immediate interrupt procedure". (This is orthogonal to the TENEX distinction between immediate and deferred TTY interrupts.)

To use interrupts a program must first tell Sail what procedure(s) to run when an interrupt happens. The routines INTMAP and PSIMAP perform this task. Deferred interrupts use the Sail process machinery (page 104), so INTSET is used to sprout the interrupt process. Then the operating system must be told to activate (and deactivate) interrupts for the desired conditions. ENABLE and DISABLE are used by the program to tell Sail, which tells the operating system.

A good knowledge of the interrupt structure of the operating system which you are trying to use should be considered a prerequisite for this chapter.

## 19.2 Interrupt Routines

### ATI, DTI

ATI (PSICHAN, CODE);
DTI (PSICHAN, CODE)

(TENEX only.) CODE is associated or dissociated with PSICHAN, using the appropriate JSYS. Executing ATI is an additional step (beyond ENABLE) which is necessary to receive TENEX TTY interrupts.

### DFR1IN

DFR1IN (AOBJN_PTR)

DFR1IN is the procedure used by DFRINT to record the interrupt and the AOBJN_PTR. Thus DFRINT is (partially) equivalent to

```
SIMPLE PROCEDURE DFRINT; BEGIN
DFR1IN (<AOBJN_PTR specified
   to INTMAP>) END;
```

To have more than one procedure run (deferred) as the result of an interrupt, a program may use DFR1IN to record the AOBJN_PTRs explicitly. Example:

```
SIMPLE PROCEDURE ZORCH;
   BEGIN
   DFRIIN (<AOBJN pointer for FOO call>);
   DFRIIN (<AOBJN pointer for BAZ call>);
   END;
...
INTMAP (INTTTY_INX, ZORCH, 0);
ENABLE (INTTTY_INX);
```

Both FOO and BAZ will be run (deferred) as the result of INTTTY_INX interrupt.

## DFRINT

DFRINT

DFRINT is a predeclared simple procedure which handles the queueing of deferred interrupts. Specify DFRINT to INTMAP for each interrupt which will be run as a Sail deferred interrupt. When run as the result of an interrupt, DFRINT grabs the AOBJN_PTR pointer specified to INTMAP (or PSIMAP) and copies the block along with other useful information into the circular deferred interrupt buffer. (See INTTBL.) DFRINT then changes the status of the interrupt process INTPRO from suspend to ready, and turns on the global integer INTRPT.

## DISABLE, ENABLE

DISABLE (INDEX);
ENABLE (INDEX)

Sail tells the operating system to ignore (DISABLE) or to send to the program (ENABLE) interrupts for the condition specified by INDEX. INDEX is a bit number (0-35) which varies from system to system; consult [SysCall]. INDEX is sometimes called a "PSI channel" on TENEX.

## INTMAP

INTMAP (INDEX, PROC, AOBJN_PTR)

(TENEX users should see PSIMAP.) The routine INTMAP specifies that the simple procedure PROC is to be run whenever the Sail interrupt handler receives an interrupt corresponding to the condition specified by INDEX. A separate INTMAP must be executed for each interrupt condition. If the same INDEX is specified on two calls to INTMAP then the most recent call is the one in effect. PROC must be a simple procedure with no formal parameters. If PROC is a user procedure then PROC is run as a Sail *immediate interrupt*.

AOBJN_PTR should be zero unless DFRINT is specified for PROC. If PROC is DFRINT (and thus will be a Sail deferred interrupt) then AOBJN_PTR gives the length and location of a block of memory describing a procedure call. Such a block has the form

```
<number of words in the block>
<1st parameter to the procedure>
<second parameter to the procedure>
...
<last parameter to the procedure>
-1,,<address of the procedure>
```

and an AOBJN_PTR to it has the form

```
-<number of words>,,<starting address>.
```

Here is an example in which FOO (I, J, K) is to be called as a deferred interrupt.

```
PROCEDURE FOO (INTEGER i, j, k); ...;
...
SAFE INTEGER ARRAY FOOBLK [1:5];
ITEMVAR IPRO; COMMENT for process item of INTPRO;
...
FOOBLK [1] ← 5;
FOOBLK [2] ← I;
FOOBLK [3] ← J;
FOOBLK [4] ← K;
FOOBLK [5] ← (-1 LSH 18)+LOCATION (FOO);
...
INTSET (IPRO ← NEW, 0); COMMENT sprout INTPRO;
INTMAP (INTTTI_INX, DFRINT,
  (-5 LSH 18) + LOCATION (FOOBLK[1]));
ENABLE(INTTTI_INX)
```

NOTE: The procedure (FOO in this case) must not be declared inside any process except the main program. Otherwise, its environment will not be available when INTPRO runs. However, there is a rather complex way to get around this by using <environment>,,PDA as the last word of the calling block. See a Sail hacker if you must do this and don't know what <environment> or PDA mean.

### INTSET

INTSET (ITM, OPTIONS)

INTSET sprouts the interrupt process INTPRO with process options OPTIONS; see page 104. The default priority of INTPRO is zero; this is the highest possible priority and no other process may have priority zero. Thus INTPRO is sure to be run first at any polling point. ITM must be an item; it will become the process item of INTPRO, the interrupt process. INTSET must be called before any deferred interrupts are used.

### INTTBL

INTTBL (NEW_SIZE)

The buffer used to queue deferred interrupts is initially 128 locations long. The queue has not been know to overflow except for programs which do not POLL very often. INTTBL changes the buffer size to NEW_SIZE. Do not call INTTBL if there are any deferred interrupts pending; wait until they have all been executed.

### PSIMAP

PSIMAP (PSICHAN, PROC,
AOBJN_PTR, LEVEL)

(TENEX only.) This routine is the same as INTMAP except that LEVEL may be specified. ROUTINE is executed at interrupt level LEVEL. (TENEX INTMAP is equivalent to PSIMAP ( , , ,3).) PROC and AOBJN_PTR have the same meaning as for INTMAP.

## 19.3 Immediate Interrupts

Do not access, create, or destroy strings, records, arrays, sets, or lists. If these data structures are needed then use deferred interrupts.

To set up an immediate interrupt say

```
INTMAP (<index>, <simple procedure name>, 0),
ENABLE (<index>)
    or on TENEX,
PSIMAP (<PSIchan>, <simple procedure name>, 0, <PSIlev>),
ENABLE (<PSIchan>)
```

where <index> is a code for the interrupt condition. To turn off an interrupt use

```
DISABLE (<index>)
```

The system will not provide user interrupts for the specified condition until another ENABLE statement is executed.

IN SUAI Sail
A procedure specified by an INTMAP statement will be executed at user interrupt level. A program operating in this mode will not be interrupted, but must finish whatever it is doing within 8/60 ths of a second. It may not do any UUOs that can cause it to be rescheduled. Also, the accumulators will not be the same ones as those that were in use by the regular program. Certain locations are set up as follows:

ACs 1-6 Set up by the system as in [Frost].

AC '15 (USER) Address of the Sail user table.

AC '16 (SP) A temporary string push down stack pointer (for the foolhardy who chose to disregard the warnings about strings in immediate interrupts).

AC '17 (P) A temporary push down stack pointer.

XJBCNI (declared in SYS:PROCES.DEF as an external integer.) Bit mask with a bit on corresponding to the current condition.

XJBTPC (declared in SYS:PROCES.DEF as an external integer.) Full PC word of regular user level program.

The interrupt will be dismissed and the user program resumed when the interrupt procedure is exited. For more information on interrupt level programming consult [Frost].

IN TOPS-10
The interrupt handler again will decode the interrupt condition and call the appropriate procedure. Since there is no "interrupt level", the interrupt procedure must not itself generate any interrupt conditions, since this will cause Sail to lose track of where in the user program it was interrupted (trapped).

Also, the Sail interrupt module sets up some temporary accumulators and JOBTPC:

AC '10 index of the interrupt condition.

AC '15 (USER) Address of the Sail user table.

AC '16 (SP) A temporary string push down list. Beware.

AC '17 (P) A temporary push down pointer.

JOBTPC (an external integer) Full PC word of regular user program.

The "real" acs -- i.e., the values of all accumulators at the time the trap occurred -- are stored in locations APRACS to APRACS+17. Thus you can get at the value of accumulator x by declaring APRACS as an external integer and referring to MEMORY [LOCATION (APRACS)+x]. When the interrupt procedure is exited the acs are restored from APRACS to APRACS+17 and the Sail interrupt handler jumps to the location stored in JOBTPC (which was set by the operating system to the location at which the trap occurred). Thus, if you want to transfer control to some location in your user program, a way to do it is to have an interrupt routine like:

```
SIMPLE PROCEDURE IROUT;
  BEGIN
  EXTERNAL INTEGER JOBTPC;
  .
  JOBTPC←LOCATION (GTFOO);
  COMMENT GTFOO is a non-simple procedure
     that contains a GO TO FOO, where FOO
     is the location to which control
     is to be passed  This allows the
     "go to solver" to be called and clean
     up any unwanted procedure activations.;
  END;
```

WARNING: this does not work very well if you were interrupted at a bad time.

IN TENEX Sail
Sail initialization does a SIR, setting up the tables to external integers LEVTAB and CHNTAB, then an EIR to turn on the interrupt system. PSIMAP fills the appropriate CHNTAB location with XWD LEV,LEVROU, where LEVROU is the address of the routine that handles the interrupts for level LEV. LEVROU saves the accumulators in blocks PS1ACS, PS2ACS, and PS3ACS, which are external integers, for levels 1 through 3 respectively. Thus for a level 3 interrupt accumulator x can be accessed by MEMORY [LOCATION (PS3ACS) + x]. The PC can be obtained by reading the LEVTAB address with the RIR JSYS. Temporary stacks are set up for both immediate and deferred interrupts.

See page 79 for an example of TENEX immediate interrupts. The functions GTRPW, RTIW, STIW provide for some of the information set up in ACs under SUAI or TOPS-10.

——————— GTRPW ———————

```
STATUS ← GTRPW (FORK);
```

The trap status of FORK is returned, using the GTRPW JSYS.

——————— RTIW, STIW ———————

```
AC1 ← RTIW (PSICHAN, @AC2);
STIW (PSICHAN, AC2, AC3);
```

The indicated JSYS is performed.

## 19.4 Clock Interrupts

(This feature is currently available only in SUAI Sail and TENEX Sail.) Clock interrupts are a kind of immediate interrupt used to approximate time sharing among processes. Every time the scheduler decides to run a process it copies the procedure's time quantum (see all about quantums of processes, page 104) into the Sail user table location TIMER. Consider the following procedure, which is roughly equivalent to the one predeclared in Sail:

```
SIMPLE PROCEDURE CLKMOD;
  IF (TIMER←TIMER-1) ≤ 0 THEN INTRPT←-1;
```

To time share several ready processes one should include polling points in the relevant process procedures and should execute the following statements:

```
INTMAP (INTCLK_INX, CLKMOD, 0);
ENABLE (INTCLK_INX);
    or on TENEX
PSIMAP (1, CLKMOD, 0, 3);
ENABLE (1);
PSIDISMS (1, 1000/60);
```

The macro SCHEDULE_ON_CLOCK_INTERRUPTS defined in ⊂SUAI⊃SYS:PROCES.DEF is equivalent to these statements. When the time quantum of a process is exceeded by the number of clock ticks since it began to run, the integer INTRPT is set, and this causes the next polling point in the process to cause a rescheduling (see about rescheduling and INTRPT on page 107). The current running process will be made ready, and the scheduling algorithm chooses a ready process to run.

In TENEX Sail clock interrupts are handled differently. Since TENEX does not directly provide for interrupting user processes on clock ticks, an inferior fork is created which periodically generates the interrupts.

### PSIDISMS

PSIDISMS (PSICHAN, MSTIME)

An inferior fork is created which interrupts the current fork every MSTIME milliseconds of real time. The inferior is approximately

```
WAIT:   MOVE    1,MSTIME      ;HOW LONG
        DISMS                 ;GO AWAY
        MOVEI   1,-1          ;HANDLE TO SUPERIOR
        MOVE    2,[bit mask]  ;SELECTED CHANNEL
        IIC                   ;CAUSE AN INTERRUPT
        JRST    WAIT          ;CONTINUE
```

### PSIRUNTM

PSIRUNTM (PSICHAN, MSTIME)

The current fork is interrupted every MSTIME milliseconds of runtime. The inferior is approximately

```
WAIT:   MOVE    1,MSTIME      ;HOW LONG
        DISMS
        MOVEI   1,-1          ;SUPERIOR FORK
        RUNTM                 ;RUNTIME OF SUPERIOR
        CAMGE   1,NEXTTIME    ;READY?
         JRST WAIT            ;NO
        ADD     1,MSTIME
        MOVEM   1,NEXTTIME    ;RECHARGE
        MOVEI   1,-1          ;SUPERIOR
        MOVE    2,[bit mask]  ;SELECTED CHANNEL
        IIC                   ;CAUSE INTERRUPT
        JRST    WAIT
```

### KPSITIME

KPSITIME (PSICHAN)

Discontinues clock interrupts on PSICHAN.

Several channels can be interrupted by PSIRUNTM or PSIDISMS, each with different timing interval.

## 19.5 Deferred Interrupts

Deferred interrupts use the Sail Process machinery (page 104) to synchronize the Sail runtime system with the running of user procedures in response to interrupts. The routine INTSET sprouts the interrupt process INTPRO, the process which eventually does the calling of deferred interrupt procedures. This process is special because it is (ordinarily) guaranteed to be the first process run after a rescheduling. (See page 107 and page 109 for information on rescheduling.) When DFRINT runs as the result of an interrupt, it copies the calling block (specified to INTMAP with the AOBJN_PTR) into the deferred interrupt buffer and turns on the global integer INTRPT. At the next polling point the process supervisor will suspend the current process and run INTPRO. INTPRO calls the procedures specified by the calling blocks in the deferred interrupt buffer, turns off INTRPT, and suspends itself. The process scheduler then runs the process of highest priority.

One very common use of deferred interrupts is

to cause an event soon after some asynchronous condition (say, TTY activation) occurs. This effect may be obtained by the following sequence:

```
INTSET (IPRO←NEW, 0); COMMENT this will cause
    the interrupt process to be sprouted and
    assigned to IPRO. This process will execute
    procedure INTPRO and will have priority zero
    (the highest possible);

INTMAP (<index>, DFRINT,
  DFCPKT (0, <event type>, <event notice>,
          <cause options>));

ENABLE (<index>);
```

In <SUAI>SYS:PROCES.DEF is the useful macro

```
DEFERRED_CAUSE_ON_INTERRUPT (<index>,
   <event type>, <notice>, <options>)
```

which may be used to replace the INTMAP statement.

The following program illustrates how deferred interrupts on TENEX can be accomplished.

```
BEGIN REQUIRE 1 NEW_ITEM;
ITEMVAR IPRO; COMMENT for process item;

PROCEDURE FOO (INTEGER I, J);
PRINT ("HI ", I, " ", J);

INTEGER ARRAY FOOBLK[1:4];
FOOBLK[1] ← 4;   COMMENT # words;
FOOBLK[2] ← 12;  COMMENT arguments;
FOOBLK[3] ← 13;
FOOBLK[4] ← -1 LSH 18 + LOCATION (FOO);

INTSET (IPRO ← NEW, 0);
PSIMAP (1, DFRINT,
   -4 LSH 18 + LOCATION (FOOBLK[1]), 3);
ENABLE (1); ATI (1, "Q"-'100);

DO BEGIN OUTCHR (".");  POLL; END UNTIL FALSE;
END;
```

The program prints dots, interspersed with "HI 12 13" for each control-Q typed on the console. Whenever a control-Q is typed, DFRINT buffers the request and makes INTPRO ready to run; then DFRINT DEBRKs (in the sense of the DEBRK JSYS) back to the interrupted code. At Sail user level the POLL statement causes the process scheduler to run INTPRO, where the deferred interrupt calling block (which was copied by DFRINT) is used to call FOO.

**THE DEFERRED INTERRUPT PROCESS - INTPRO**

INTPRO first restores the following information which was stored by DFRINT at the time of the interrupt.

| LOCATION | CONTENTS |
|---|---|
| USER | The base of the user table (GOGTAB). |
| AC 1 | Status of spacewar buttons. |
| AC 2 | Your job status word (JBTSTS). See [Frost]. |
| IJBCNI(USER) | XJBCNI (i.e., JOBCNI) at time of interrupt. |
| IJBTPC(USER) | XJBTPC (i.e., JOBTPC) at time of interrupt. |
| IRUNNR(USER) | Item number of running process at time of interrupt. |

Then INTPRO calles the procedure described by the calling block. When the procedure is finished, INTPRO looks to see if the deferred interrupt buffer has any more entries left. If it does, INTPRO handles them in the same manner. Otherwise INTPRO suspends itself and the highest priority ready process takes over.

# SECTION 20

# LEAP RUNTIMES

We will follow the same conventions for describing Leap execution time routines as were used in describing the runtimes of the Algol section of Sail (see page 33).

## 20.1 Types and Type Conversion

### TYPEIT

CODE ← TYPEIT (ITM);

The type of the datum linked to an item is called the type of an item. An item without a datum is called untyped. TYPEIT is an integer function which returns an integer CODE for the type of the item expression ITM that is its argument. The codes are:

0 - item deleted or never allocated
1 - untyped
2 - Bracketed Triple item
3 - string
4 - real
5 - integer
6 - set
7 - list
8 - procedure item
9 - process item
10 - event item
11 - context item
12 - reference item
13 - record pointer
14 - label
15 - record class
23 - string array
24 - real array
25 - integer array
26 - set array
27 - list array
31 - context array
33 - record pointer array
37 - error (the runtime screwed up)

The user is encouraged to use TYPEIT. It requires the execution of only a few machine instructions and can save considerable debugging time.

### CVSET

SET ← CVSET (LIST)

CVSET returns a set given a list expression by removing duplicate occurrences of items in the list, and reordering the items into the order of their internal integer representations.

### CVLIST

LIST ← CVLIST (SET)

CVLIST returns a list given a set expression. It executes no machine instructions, but merely lets you get around Sail type checking at compile time.

### CVN and CVI

INTEGR ← CVN (ITM);
ITM ← CVI (INTEGR)

CVN returns the integer that is the internal representation of the item that is the the value of the item expression ITM. CVI returns the item that is represented by the integer expression INTEGR that is its argument. Legal item numbers are between (inclusively) 1 and 4095, but you'll get in trouble if you CVI when no item has been created with that integer as its representation. Absolutely no error checking is done. CVI is for daring men. See about item implementation, page 86, for more information about the internal representations of items.

### MKEVTT

MKEVTT (ITEM)

MKEVTT will convert its item argument to an event type item. The old datum will be overwritten. The type of the item will now be "event type". Any item except an event type item may be converted to an event type item by MKEVTT.

## 20.2 Make and Erase Breakpoints

### BRKERS, BRKMAK, BRKOFF

BRKMAK (BREAKPT_PROC);
BRKERS (BREAKPT_PROC);
BRKOFF

In order to give the programmer some idea of what is going on in the associative store, there is a provision to interrupt each MAKE and ERASE operation, and enter a breakpoint procedure. The user can then do whatever he wants with the three items of the association being created or destroyed. ERASE Foo ⊛ ANY ≡ ANY will cause the breakpoint procedure to be activated once for each association that matches the pattern. MAKE it1 ⊛ it2 ≡[it3 ⊛ it4 ≡ it5] will cause the breakpoint procedure to be activated twice.

The user's breakpoint procedures must have the form:

```
PROCEDURE Breakpt_proc (ITEMVAR a, o, v )
```

If the association being made or erased is A⊛O≡V, then directly before doing the Make or Erase, Breakpt_proc is called with the items A, O, and V for the formals a, o, and v.

To make the procedure Breakpt_proc into a breakpoint procedure for MAKE, call BRKMAK with Breakpt_proc as a parameter. To make the procedure Breakpt_proc into a breakpoint procedure for ERASE, call BRKERS with Breakpt_proc as its parameter. To turn off both breakpoint procedures, call BRKOFF with no parameters.

NOTE: BRKMAK, BRKERS and BRKOFF are not predeclared. The user must include the declarations:

```
EXTERNAL PROCEDURE BRKERS (PROCEDURE BP);
EXTERNAL PROCEDURE BRKMAK (PROCEDURE BP);
EXTERNAL PROCEDURE BRKOFF
```

## 20.3 Pname Runtimes

### CVIS

"PNAME" ← CVIS (ITEM, @FLAG)

*The print name of ITEM* is returned as a string. Items have print names only if one includes a REQUIRE n PNAMES statement in his program, where n is an estimate of the number of pnames the program will use. An Item's print name is the identifier used to declare it, or that pname explicitly given it by the NEW_PNAME function (see below). FLAG is set to False (0) if the appropriate string is found. Otherwise it is set to TRUE (-1), and one should not put great faith in the string result.

### CVSI

ITEM ← CVSI ("PNAME", @FLAG)

The Item whose pname is the same as the string *argument PNAME is returned* and FLAG is set to FALSE if such an ITEM exists. Otherwise, something very random is returned, and FLAG is set to TRUE.

### DEL_PNAME

DEL_PNAME (ITEM)

This function deletes any string PNAME associates with this ITEM.

### NEW_PNAME

NEW_PNAME (ITEM, "STRING")

This function assigns to the Item the name "STRING". Don't perform this twice for the same Item without first deleting the previous one. The corresponding name or Item may be retrieved using CVIS or CVSI (see above). The NULL string is prohibited as the second argument.

## 20.4 Other Useful Runtimes

### LISTX

VALUE ← LISTX (LIST, ITEM, N)

The value of this integer function is 0 if the ITEM (an item expression) does not occur in the list at least N (an integer expression) different times in the LIST (a list expression). Otherwise LISTX is the index of the Nth occurrence of ITEM in LIST. For example,

```
LISTX ({{Foo, Baz, Garp, Baz}}, Baz, 2) is 4.
```

### FIRST, SECOND, THIRD

ITEM ← FIRST (BRAC_TRIP_ITEM);
ITEM ← SECOND (BRAC_TRIP_ITEM);
ITEM ← THIRD (BRAC_TRIP_ITEM)

The Item which is the FIRST, SECOND, or THIRD element of the association connected to a bracketed triple item (BRAC_TRIP_ITEM) is returned. If the item expression BRAC_TRIP_ITEM does not evaluate to a bracketed triple, an error messages issues forth.

### ISTRIPLE

RSLT ← ISTRIPLE (ITM)

If ITM is a bracketed triple item then ISTRIPLE returns TRUE; otherwise it returns FALSE. ISTRIPLE (ITM) is equivalent to (TYPEIT (ITM) = 2).

### LOP

ITEM ← LOP (@SETVARIABLE);
ITEM ← LOP (@LISTVARIABLE)

LOP will remove the first item of a set or list from the set or list, and return that item as its value. Note that the argument must be a variable because the contents of the set or list is changed. If one LOPs an empty set or a null list, an error message will be issued.

### COP

ITEM ← COP (SETEXPR);
ITEM ← COP (LISTEXPR)

COP will return the first item of the set or list just as LOP (above) will. However, it will NOT remove that item from the set or list. Since the set or list will be unchanged, COP's argument may be a set or list expression. As with LOP, an error message will be returned if one COPs an empty set or a null list.

### LENGTH

VALUE ← LENGTH (SETEXPR);
VALUE ← LENGTH (LISTEXPR)

LENGTH will return the number of items in that set or list that is its argument. LENGTH (S) = 0 is a much faster test for the null set or list that S = PHI or S = NIL.

### SAMEIV

VALUE ← SAMEIV (ITMVAR1, ITMVAR2)

SAMEIV is useful in Matching Procedures to solve a particular problem that arises when a Matching Procedure has at least two ? itemvar arguments. An example will demonstrate the problem:

```
FOREACH X | Matchingproc ( X, X ) DO ...;
FOREACH X, Y | Matchingproc ( X, Y ) DO ... ;
```

Clearly, the matching procedure with both arguments the same may want to do something different from the matching procedure with two different Foreach itemvars as its arguments. However, there is no way inside the body of the matching procedure to differentiate the two cases since in both cases both itemvar formals have the value BINDIT. SAMEIV will return True only in the first case, namely 1) both of its arguments are ? itemvar formals to a matching procedure, 2) both had the same Foreach itemvar passed by reference to them. It will return False under all other conditions, including the case where the Foreach itemvar is bound at the time of the call (so it is not passed by reference, but its item value is passed by value to both formals).

## 20.5 Runtimes for User Cause and Interrogate Procedures

### SETCP AND SETIP

SETCP (ETYPE, PROC_NAME);
SETCP (ETYPE, DATUM (PROC_ITEM));
SETIP (ETYPE, PROC_NAME);
SETIP (ETYPE, DATUM (PROC_ITEM))

SETCP and SETIP associate with the event type specified by the item expression ETYPE a procedure specified by its name or the datum of a procedure item expression.

After the SETCP, whenever a Cause statement of the specified event type is executed, the procedure specified by PROC_NAME or PROC_ITEM is called. The procedure must have three formal parameters corresponding to the event type, event notice, and options words of the CAUSE statement. For example,

```
PROCEDURE CAUSEIT (ITEMVAR ETYP, ENOT;
    INTEGER OP)
```

After SETIP, whenever an Interrogate statement of the specified event type is executed, the procedure specified by PROC_NAME or PROC_ITEM is called. The procedure must have two formal parameters corresponding to the event type and options words of the Interrogate statement and return an item. For example,

```
ITEM PROCEDURE ASK_IT (ITEMVAR ETYP;
    INTEGER OP)
```

It is an error if a Cause or Interrogate statement tries to call a procedure whose environment (static - as determined by position of its declaration, and dynamic - as determined by the execution of the SETCP or SETIP) has been exited.

See page 112 and page 113 for more information on the use of SETCP and SETIP, respectively.

### CAUSE1

ITMVAR ← CAUSE1 (ETYPE, ENOT, OPTIONS);
ITMVAR ← CAUSE1 (ETYPE, ENOT);
ITMVAR ← CAUSE1 (ETYPE)

CAUSE1 is essentially the procedure executed for CAUSE statements if no SETCP has been done for the event type ETYPE. See the description of the Sail defined Cause statement, page 112, for further elucidation.

### ASKNTC

ITMVR ← ASKNTC (ETYPE ,OPTIONS);
ITMVR ← ASKNTC (ETYPE)

ASKNTC is the procedure executed for INTERROGATE statements if no SETIP has been done for the event type ETYPE. See the description of the Sail defined Interrogate statement, page 113, for further elucidation.

### ANSWER

BITS ← ANSWER (ETYPE, ENOT, PROC_ITEM)

ANSWER will attempt to wake up from an interrogate wait the process specified by the item expression PROC_ITEM. If the process is not in a suspended state, Answer will return an integer with the bit '400000 in the right half (NOJOY in ⊂SUAI⊃SYS:PROCES.DEF) turned on. If the process is suspended, it will be made ready, and removed from any wait queues it may be on. The bits corresponding to the options word of the interrogate statement that put it in a wait state will be returned. Furthermore, if the SAY_WHICH bit was on, the appropriate association, namely EVENT_TYPE ⊗ ENOT ≡ ETYPE, will be made. See page 112 for more information on the use of ANSWER.

### DFCPKT

AOBJN_PTR ← DFCPKT (@BLOCK, EVTYP,
EVNOT, OPTS)

This routine is a convenience for causing an event as a deferred interrupt. If BLOCK is non-

zero then it should be an array with at least 5 elements; if BLOCK is zero then a five-word block is allocated. DFCPKT constructs a call for CAUSE (EVTYP, EVNOT, OPTS) in this block and returns an AOBJN pointer to it.

# SECTION 21

# BASIC CONSTRUCTS

## 21.1 Syntax

```
<variable>
      ::= <identifier>
      ::= <identifier> [ <subscript_list> ]
      ::= DATUM ( <typed_item_expression> )
      ::= DATUM ( <typed_item_expression> ) [
            <subscript_list> ]
      ::= PROPS ( <item_expression> )
      ::= <context_element>
      ::= <record_class> : <field> [
            <record_pointer_expression> ]


<typed_item_expression>
      ::= <typed_itemvar>
      ::= <typed_item>
      ::= <typed_itemvar_procedure>
      ::= <typed_item_procedure>
      ::= <typed_itemvar_array>
            [ <subscript_list> ]
      ::= <typed_item_array>
            [ <subscript_list> ]
      ::= <itemvar> ← <typed_item_expression>
      ::= IF <boolean_expression> THEN
            <typed_item_expression> ELSE
            <typed_item_expression>
      ::= CASE <algebraic_expression> OF (
            <typed_item_expression_list> )


<typed_item_expression_list>
      ::= <typed_item_expression>
      ::= <typed_item_expression_list> ,
            <type_item_expression>

<subscript_list>
      ::= <algebraic_expression>
      ::= <subscript_list> ,
            <algebraic_expression>
```

## 21.2 Semantics

VARIABLES

If a variable is simply an identifier, it represents a single value of the type given in its declaration.

If it is an identifier qualified by a subscript list it represents an element from the array bearing the name of the identifier. However, an identifier qualified by a subscript list containing only a single subscript may be either an element from a one dimensional array, or an element of a list. Note that the token "∞" may be used in the subscript expression of a list to stand for the length of the list, e.g. LISTVAR[∞-2]←LISTVAR[∞-1].

The array should contain as many dimensions as there are elements in the subscript list. A[I] represents the I+1th element of the vector A (if the vector has a lower bound of 0). B[I, J] is the element from the I+1th row and J+1th column of the two-dimensional array B. To explain the indexing scheme precisely, all arrays behave as if each dimension had its origin at 0, with (integral) indices extending infinitely far in either direction. However, only the part of an array between (and including) the lower and upper bounds given in the declaration are available for use (and in fact, these are the only parts allocated). If the array is not declared SAFE, each subscript is tested against the bounds for its dimension. If it is outside its range, a fatal message is printed identifying the array and subscript position at fault. SAFE arrays are not bounds-checked. Users must take the consequences of the journeys of errant subscripts for SAFE arrays. The bounds checking causes at least three extra machine instructions (two of which are always executed for valid subscripts) to be added for each subscript in each array reference. The algebraic expressions for lower and upper bounds in array declarations, and for subscripts in subscripted variables, are always converted to integer values (see page 23) before use.

For more information about the implementation of Sail arrays, see page 157.

DATUMS

DATUM (X) where X is a typed item expression, will act exactly like a variable with the type of the item expression. The programmer is

responsible for seeing that the type of the item is that which the DATUM construct thinks it is. For example, the Datum of a Real Itemvar will always interpret the contents of the Datum location as a floating point number even if the program has assigned a string item to the Real Itemvar.

PROPS
The PROPS of an item will always act as an integer variable. Any algebraic value assigned to a props will be coerced to an integer (see about type conversions, page 23) then the low order 12 bits will be stored in the props of the item. Thus, the value returned from a props will always be a non-negative integer less than '7777 (4095 in decimal).

RECORD FIELDS
A field in a record is also a variable. The variable is allocated and deallocated with the other fields of the same record as the result of calls to NEW_RECORD and the record garbage collector. For more information see page 65.

IDENTIFIERS
You will notice that no syntax was included for the non-terminal symbols <identifier> or <constant>. It is far easier to explain these constructs in an informal manner.

A Sail letter is any of the upper or lower case letters A through Z, or the underline character (_ or !, they are treated equivalently). Lower case letters are mapped into the corresponding upper case letters for purposes of symbol table comparisons (SCHLUFF is the same symbol as Schluff). A digit is any of the characters 0 through 9.

An identifier is a string of characters consisting of a letter followed by virtually any number of letters and digits There must be a character which is neither a letter nor a digit (nor either of the characters "." or "$") both before and after every identifier. In other words, if YOU can't determine where one identifier ends and another begins in a program you have never seen before, well, neither can Sail.

There is a set of identifiers which are used as Sail delimiters (in the Algol sense -- that is, BEGIN is treated by Algol as if it were a single character; such an approach is not practical, so a reserved identifier is used). These identifiers are called Reserved Words and may not be used for any purpose other than those given explicitly in the syntax, or in declarations (DEFINES) which mask their reserved-word status over the scope of the declarations. E.g., "INTEGER BEGIN" is allowed, but a Synonym (see page 10) should have been provided for BEGIN if any new blocks are desired within this one, because BEGIN is ONLY an Integer in this block. Another set of identifiers have preset declarations -- these are the execution time functions. These latter identifiers may also be redefined by the user; they behave as if they were declared in a block surrounding the outer block. A list of reserved words and predeclared identifiers may be found in the appendices. It should be noted that due to the stupidity of the parser, it is impossible to declare certain reserved words to be identifiers. For example, INTEGER REAL; will give one the syntax error "Bogus token in declaration".

Some of the reserved words are equivalent to certain special characters (e.g. "|" for "SUCH THAT"). A table of these equivalences may be found in the appendices.

ARITHMETIC CONSTANTS

| | |
|---|---|
| 12369 | Integer with decimal value 12369 |
| '12357 | Integer with octal value 12357 |
| 123. | Real with floating point value 123.0 |
| 0123.0 | Real with floating point value 123.0 |
| .524 | Real with floating point value 0.524 |
| 5.3@2 | Real with floating point value 530.0 |
| 5.342@-3 | Real with floating point value 0.005342 |

The character ' (right quote) precedes a string of digits to be converted into an OCTAL number.

If a . or a @ appears in a numeric constant, the type of the constant is returned as Real (even if it has an integral value). Otherwise it is an integer. Type conversions are made at compile time to make the type of a constant commensurate with that required by a given operation. Expressions involving only constants are evaluated by the compiler and the resultant values are substituted for the expressions.

The reserved word TRUE is equivalent to the Integer (Boolean) constant -1; FALSE is equivalent to the constant 0.

## STRING CONSTANTS

A String constant is a string of ASCII characters (any which you can get into a text file) delimited at each end by the character ". If the " character is desired in the string, insert two " characters (*after* the initial delimiting " character, of course).

A String constant behaves *like any other* (algebraic) primary. It is originally of type String, but may be converted to Integer by extracting the first character if necessary (see page 23).

The reserved word NULL represents a String constant containing no characters (length=0).

Examples: The left hand column in the table that follows gives the required input

| INPUT | RESULT | LENGTH |
|---|---|---|
| "A STRING" | A STRING | 8 |
| "WHAT'S ""DOK"" MEAN?" | WHAT'S "DOK" MEAN? | 18 |
| """A QUOTED STRING""" | "A QUOTED STRING" | 17 |
| "" | | 0 |
| NULL | | 0 |

## COMMENTS

If the scanner detects the identifier COMMENT, all characters up to and including the next *semicolon* (;) will be ignored. A comment may appear anywhere as long as the word COMMENT is properly delimited (not in a String constant, of course);

A string constant appearing just before a statement also has the effect of a comment.

# SECTION 22

# USING SAIL

## 22.1 For TOPS-10 Beginners

If you simply want your Sail program compiled, loaded, and executed, do the following:

1. Create a file called "XXXXXX.SAI" with your program in it, where "XXXXXX" may be any name you wish.

2. Get your job to monitor level and type "EXECUTE XXXXXX".

3. The system program (variously called SNAIL, COMPILE, RPG) which handles requests like EXECUTE will then start Sail. Sail will say "Sail: XXXXXX". When Sail hits a page boundary in your file, it will type "1" or whatever the number of the page that it is starting to read.

4. When the compilation is complete Sail swaps to the loader, which will say "LOADING".

5. When the loading is complete the loader will type "LOADER nP CORE" where n is your core size. The loader then says "EXECUTION".

6. When execution is complete Sail will type "End of Sail execution" and exit.

At any time during 3 through 6 above, you could get an error message from Sail of the form "DRYROT: <cryptic text>", or from the system, such as "ILL MEM REF", "ILLEGAL UUO" etc. followed by some core locations. These are Sail bugs. You will have to see a Sail hacker about them, or attempt to avoid them by rewriting the offending part of your program, or try again tomorrow.

If you misspell the name of your file then SNAIL will complain "File not found: YYYYYY" where "YYYYYY" is your misspelling. Otherwise, the error messages you receive during 3 above will be compilation errors (bad syntax, type mismatch, begin-end mismatch, unknown identifiers, etc.). See page 138 about these.

If you get through compilation (step 3) with no error messages, the loading of your program will rarely fail. If it somehow does, it will tell you. See a Sail hacker about these.

If you also get through loading (step 4) with no errors, you aren't yet safe. Sail will give you error messages during the execution of your program if you exceed the bounds of an array, refer to a field of a null record, etc. See section 1 about these too.

If you never get an error message, and yet you don't get the results you thought you'd get, then you've probably made some mistakes in your programming. Use BAIL (or RAID or DDT) and section 2 to aid in debugging. It is quite rare for Sail to have compiled runable but incorrect code from a correct program. The only way to ascertain whether this is the case is to isolate the section of your program that is causing Sail to generate the bad code, and then patiently step through it instruction by instruction using RAID or DDT, and check to see that everything it does makes sense.

## 22.2 For TENEX Beginners

If you simply want your Sail program compiled, loaded, and executed, do the following.

1. Create a file called "XXXXXX.SAI" with your program in it, where XXXXXX may be any name you wish.

2. Type "Sail", followed by a carriage return, to the TENEX EXEC.

3. The EXEC will load and start Sail. Sail will say "Tenex Sail 8.1 8-5-76 *". Type "XXXXXX<cr>" (your file name). Sail will create a file XXXXXX.REL, and will type the page number of the source file as it begins to compile each page.

4. When Sail finishes it will type "End of compilation.". Return to the EXEC and type "LOADER<cr>". The loader will type "*". Type

"SYS:LOWTSA,DSK:XXXXXX$", where $ is the altmode key. This loads your program into core.

5. When the LOADER exits, the program is loaded. You may now either SAVE the program, for later use, or run it with the EXEC START command.

## 22.3 The Complete use of Sail

The general sequence of events in using Sail is:

1. Start Sail.
2. Compile one or more files into one or more binary files, with possibly a listing file generated.
3. Load the binary file(s) with the appropriate upper segment or with the Sail runtime library, and possibly with RAID or DDT.
4. Start the program, possibly under the control of BAIL, RAID or DDT.
5. Let the program finish, or stop it to use a debugger or to reallocate storage with the REENTER command.

Starting Sail is automatic with the SNAIL commands described below. Otherwise, "R SAIL" will do.

## 22.4 Compiling Sail Programs

When started explicitly by monitor command, Sail will type back an "*" at you and wait for you to type in a <command line>. It will do the compilation specified by that command line, then ask for another, etc.

If you use SNAIL then follow the SNAIL command with a list of <command line>s separated by commas. The compilation of each <command line> will be done before the next <command line> is read and processed. The SNAIL commands are:

| | |
|---|---|
| EXecute | compile, load, start |
| TRY | compile, load with BAIL, start |
| DEBug | compile, load with BAIL, start BAIL |
| LOAd | compile, load |
| PREPare | compile, load with BAIL |
| COMpile | compile |

See [MonCom] for more information about the use of SNAIL and the switches available to it.

COMMAND LINE SYNTAX
TOPS-10 COMMAND LINE SYNTAX

```
<command_line>
     ::= <binary_name> <listing_name> ←
          <source_list>
     ::= <file_spec> @
     ::= <file_spec> !

<binary_name>
     ::= <file_spec>
     ::= <empty>

<listing_name>
     ::= , <file_spec>
     ::= <empty>

<source_list>
     ::= <file_spec>
     ::= <source_list> , <file_spec>

<file_spec>
     ::= <file_name> <file_ext> <proj_prog>
     ::= <device_name> <file_spec> <switches>
     ::= <device_name> <switches>

<file_name>
     ::= <legal_sixbit_id>

<file_ext>
     ::= . <legal_sixbit_id>
     ::= <empty>

<proj_prog>
     ::= [ <legal_sixbit_id> ,
          <legal_sixbit_id> ]
     ::= <empty>
```

```
<device_name>
     ::= <legal_sixbit_id>


<switches>
     ::= ( <unslashed_switch_list> )
     ::= <slashed_switch_list>
     ::= <empty>


<unslashed_switch_list>
     ::= <switch_spec>
     ::= <unslashed_switch_list> <switch_spec>



<slashed_switch_list>
     ::= / <switch_spec>
     ::= <slashed_switch_list> / <switch_spec>


<switch_spec>
     ::= <valid_switch_name>
     ::= <signed_integer> <valid_switch_name>


<valid_switch_name>
     ::= A
     ::= B
     ::= C
     ::= D
     ::= F
     ::= H
     ::= K
     ::= L
     ::= P
     ::= Q
     ::= R
     ::= S
     ::= V
     ::= W
     ::= X
```

## TENEX SAIL COMMAND LINE SYNTAX

```
<command_line>
     ::= <file_list> CR
     ::= <file_list> , CR
     ::= <file_list> ←
     ::= <file_list> , ←
     ::= ← <file_list>
     ::= ?


<file_list>
     ::= <file> , <file_list>


<subcommand>
     ::= CR
     ::= <control-R>
     ::= <control-L>
     ::= / <switch>
     ::= ?


<switch>
     ::= <number> <switch>
     ::= <TOPS-10 switch>
     ::= G
     ::= I
     ::= T
```

### COMMAND LINE SEMANTICS

All this is by way of saying that Sail accepts commands in essentially the same format accepted by other processors written for the operating system on which you are running. *The binary file name is the name* of the output device and file on which the ready to load object program will be written. The listing file, if included, will contain a copy of the source files with a header at the top of each page and an octal program counter entry at the head of each line (see page 134). The listing file name is often omitted (no listing created). The source file list specifies a set of user-prepared files which, when concatenated, *form a valid Sail* program (one outer block).

If file_ext is omitted from the binary_name then the extension for the output file will be .REL. The default extension for the listing file is .LST. Sail will first try to find source files under the names given. If this fails, and the extension is omitted, the same file with a .SAI extension will be tried.

If device_name is omitted then DSK: is assumed. If proj_prog is omitted, the project-*programmer number* for the job is assumed.

Switches are parameters which affect the operation of the compiler. A list of switches may appear after any file name on TOPS-10; use subcommand mode on TENEX. The parameters specified are changed immediately after the file name associated with them is processed. The meanings of the switches are given below.

The binary, listing and (first) source file names are processed before compilation -- subsequent source names (and their switches) are processed whenever an end-of-file condition is detected in the current source file. Source files which appear after the one containing the outer block's END delimiter are not ignored, but should contain only comments.

Each new line in the command file (or entered from the teletype) specifies a separate program compilation. Any number of programs can be compiled by the same Sail core image.

The file_spec@ command causes the compiler to open the specified file as the command file. Subsequent commands will come from this file. If any of these commands is file_spec@, another switch will occur.

The file_spec ! command will cause the specified file to be run as the next processor. This program will be started in "RPG mode". That is, it will look on the disk for its commmands if its standard command file is there -- otherwise, command control will revert to the TTY. The default option for this file name is .DMP. The default device is SYS.

TENEX Sail command syntax is much like the syntax of the TENEX DIRECTORY command. Filenames are obtained from the terminal using recognition; .SAI, .REL, and .LST are the default extensions. Command lines ending in comma or comma backarrow enter subcommand mode. Command lines ending in backarrow cause termination of command scanning and start compilation; the program will be loaded with DDT and DDT will be started. A file name appearing before a backarrow is taken as a source file; the .REL file will have the same (first) filename. A command line beginning with backarrow causes no .REL file to be generated. In subcommand mode the characters control-R and control-L allow complete specification of the binary and listing file names, respectively.

SWITCHES

The following table describes the Sail parameter switches. If the switch letter is preceded in the table by the D character, a decimal number is expected as an argument. 0 is the default value. The character O indicates that an octal number is expected for this switch. Otherwise the argument is ignored.

ARG SWITCH FUNCTION

O A The octal number O specifies bits which determine the code compiled in certain cases.

- 1 use KIFIX for real to integer conversion
- 2 use FIXR
  ;otherwise use UUOFIX
- 4 use FLTR for integer to real conversion
  ;otherwise use UUOFLOAT
- 10 use ADJSP whenever possible
  ;otherwise use SUB, or ADD with PDLOV detection
- 20 use FORTRAN-10 calling sequence for calling Fortran Procedures; else old F40 style

The compiler is initialized with /0A; the compiled code will run on a KA10 using F40 calling sequence for Fortran Procedures.

O B The octal number O specifies bits which determine how much information is produced for BAIL.

- 1 Program counter to source/listing directory.
- 2 Include information on all symbols. If not selected then do not include non-internal local variables.
- 4 SIMPLE procedures get proc descriptors.
- 10 Don't automatically load SYS:BAIL.REL.
- 20 Make the Sail predeclared runtimes known by requiring SYS:BAIPDn.REL.

C This switch turns on CREFfing. The listing file (which must exist) will be in a format suitable for processing by CREF, the program which will generate a cross-reference listing of your Sail program from your listing files.

D D If the decimal number D is zero or does not appear then double the amount of space for the push down stack used in expanding macros (see page 57). If D is not zero then set the stack size to D. Use this switch if the compiler indicates to you that this stack has overflowed. This shouldn't happen unless you nest DEFINE calls extremely deeply.

O F O is an octal number which specifies exactly what kind of listing format is

generated. O contains information about 7 separate listing features, each of which is assigned a bit in O.

1 List the program counter (see / L switch ).
2 List with line numbers from the source text.
4 List the macro names before expansion.
10 Expand macro texts in the listing file.
20 Surround each listed macro expansion with < and > .
40 Suspend listing.
100 No banner at the top of each page.
[This is a way to "permanently" expand macros. A /110F listing is (almost) suitable as a Sail source file.]

The compiler is initialized with /7f (i.e., list program counter, line numbers, and macro names).

G (TENEX only) Load after compilation, exiting to the monitor.

H (Default on TENEX) This switch is used to make your program sharable. When loaded, the code and constants will be placed in the second (write-protected) segment, while data areas will be allocated in the lower, non-shared segment. Programs compiled with /H request SYS:HLBSAn as a library (<SAIL>HLBSAn on TENEX). The sharable library HLBSAn is identical to LIBSAn, except that it expects to run mostly in the upper (shared) segment. Recall that n is the current version number. At SUAI, use the monitor command SETUWP to write protect the upper segment. Then SSAVE the core image.

I (TENEX only) Do not compile two-segment code.

K The counter mechanism of Sail is activated, enabling one to determine the frequency of execution of each statement in your Sail program. See Appendix F, the Statement Counter System. This switch is ignored unless a listing is specified with a /LIST.

O L In compiling a Sail program, an internal variable called PCNT (for program counter) is incremented (by one) for each word of code generated. This value, initially 0, represents the address of a word of code in the running program, relative to the load point for this program. The current octal value of PCNT plus the value of another internal variable called LSTOFFSET, is printed at the beginning of each output line in a listing file. For the first program compiled by a given Sail core image, LSTOFFSET is initially 0. If the L switch occurs in the command and the value O is non-negative, O replaces the current value of LSTOFFSET. If O is -1, the current size of DDT is put into LSTOFFSET. If O is -2, the current size of RAID is used. In "RPG mode" the final value of PCNT is added to LSTOFFSET after each compilation. Thus by deleting all .REL files produced by Sail, and by compiling all Sail programs which are to be loaded together with one RPG command which includes the L switch, you can obtain listing files such that each of these octal numbers represents the actual starting core address of the code produced by the line it precedes. At the time of this writing, SNAIL would not accept minus signs in switches to be sent to processors. Keep trying.

D P Set the size of the system pushdown list to D (decimal). If D is zero or does not appear then double the (current) size of the list. Thus /35P/P will first set the stack size to 35, then double it to 70. It has never been known to overflow.

D Q Set the size of the string pushdown list to D (decimal). If D is zero or does not appear then double the size of the list. No trouble has been encountered here, either.

D R Set the size of the compiler's parsing and semantic stacks to D (decimal). If D is zero or does not appear then double the size of the stacks. A long conditional statement of the form (IF ... THEN ... ELSE IF ... THEN ... ELSE IF ... ) has been known to

cause these stacks to overflow their normally allocated sizes.

D S The size of String space is Set to D words. String space usage is a function of the number of identifiers, especially macros, declared by the user. In the rare case of String space exhaustion, 5000 is a good first number to try.

T (TENEX only) Load with DDT, exit to DDT.

V Always put loader link blocks and the characters for constant strings into the low segment, even if /H is selected. This is intended for use in overlay systems where code is overlaid but data is not.

W Generate additional suppressed DDT symbols. These symbols are designed to serve as comments to a programmer or processor rummaging though the generated code. Symbols generated by this switch all begin with a percent sign (%), and many come in pairs. A %$ symbol points to the first word of an area and a %. symbol points to the first word beyond the area. Thus the length of an area is the difference of its %. and %$ symbols. The symbols are:

| | | |
|---|---|---|
| %$ADCN | %.ADCN | address constants |
| %$LIT | %.LIT | literals |
| %$RLIT | %.RLIT | reference literals |
| %$SCOD | %.SCOD | START(or QUICK)_CODE |
| %$STRC | %$STRC | string variables |
| %$VARS | %.VARS | simple variables |
| %ALSTO | | start to clear registers |
| %$ARRY | | first data word of a fixed array |
| %$FORE | | FOREACH satisfier block |
| %$SUCC | | SUCCEED/FAIL return block |

/W tends to increase the number of DDT symbols by a factor of 2 or 3.

X Enable compiler save/continue (page 159).

Here is an example of a compile string which a user who just has to try every bell and whistle available to him might type to compile a file named NULL:

```
COMPILE /LIST /SAIL NULL(RR-2L5000S)
```

The switch information contained in parentheses will be sent unchanged to Sail. Note the convention which allows one set of parentheses enclosing a myriad of switches to replace a "/" character inserted before each one. This string tells the compiler to compile NULL using parse and semantic stacks four times larger than usual (RR). A listing file is to be made which assumes that RAID will be loaded and NULL will be loaded right after RAID (-2L). His program is big enough to need 5000 words of String space (5000S). The statement REQUIRE "chars" COMPILER_SWITCHES; can be used to change the settings of the compiler switches. "chars" must be a string constant which is a legitimate switch string, containing none of the characters "(/)"; e.g.,

```
REQUIRE "20F" COMPILER_SWITCHES;
```

The string of characters is merely passed to the switch processor, and it may be possible to cause all sorts of problems depending on the switches you try to modify. Switches A, B, and F are the only ones usually modified. The switches which set stack sizes (D, P, Q, R) or string space (S) should be avoided. Switches which control the format of files (B, F) should only be used if you have such a file open.

## 22.5 Loading Sail Programs

Load the main program, any separately compiled procedure files (see page 12), any assembly language (see page 13) or Fortran procedures, and DDT or RAID if desired. This is all automatic if you use the LOAD or DEBUG or EXECUTE system commands (see [MonCom]). Any of the Sail execution time routines requested by your program will be searched out and loaded automatically from SYS:LIBSAn.REL (<SAIL>LIBSAn on TENEX). If

the shared segment is available and desired, type SYS:SAILOW (SYS:LOWTSA for TENEX) as as your very first LOADER command (before /D even). SUAI people can abbreviate SYS:SAILOW as /Y. All this is done automatically by SNAIL at SUAI. Other loaders (e.g., LINK10) can also be used.

## 22.6 Starting Sail Programs

For most applications, Sail programs can by started using the START, RUN, EXECUTE, or TRY system commands, or by using the SG command of DDT (RAID). The Sail storage areas will be initialized. This means that all knowledge of I/O activity, associative data structures, strings, etc. from any previous activation of the program will be lost. All strings (except constants) will be cleared to NULL. All compiled-in arrays will not be reinitialized (PRELOADed arrays are preloaded at compile time - OWN arrays are never initialized). Then execution will begin with the first statement in the outer block of your main program. As each block is entered, its arrays will be cleared as they are allocated. Variables are not cleared. The program will exit when it leaves this outer block.

STARTING THE PROGRAM IN "RPG" MODE
Sail programs may be started at one of two consecutive locations: at the address contained in the cell JOBSA in the job data area, or at the address just following that one. The global variable RPGSW is set to 0 in the former case, -1 in the latter. Aside from this, there is no difference between the two methods. This cell may be examined by declaring RPGSW as an EXTERNAL INTEGER.

## 22.7 Storage Reallocation with REEnter

The compiler dynamically allocates working storage for its push down lists, symbol tables, string spaces, etc. It normally runs with a standard allocation adequate for most programs. Switch settings given above may be used to change these allocations. If desired, these allocations may also be changed by typing ↑C, followed by REE (reenter). The compiler will ask you if you want to allocate. Type Y to allocate, N to use the standard allocation, and any other character to use the standard allocations and print out what they are. All entries will be prompted. Numbers should be decimal. Typing alt-mode instead of CR will cause standard allocation to be used for the remaining values. The compiler will then start, awaiting command input from the teletype.

For SUAI "Global Model" users, the REE command will also delete any REQUIREd or previously typed segment name information. The initialization sequence will then ask for new names.

# SECTION 23

# DEBUGGING SAIL PROGRAMS

## 23.1 Error Messages

If the compiler detects a syntax or semantic error while compiling a program it will provide the user with the following information:

1) The error message. These are English phrases or sentences which attempt to diagnose the problem. If a message is vague it is because no specific test for the error has been made and a catchall routine detected it. If the message begins with the word "DRYROT" it means that there is a bug in the compiler which some strangeness in your program was able to tickle. See a system programmer about this.

2) The current input line. Page and line number, along with the text of the line being scanned, are typed. A line feed will occur at the point in the line just following the last program element scanned. The absence of a position indicator means that a macro (DEFINE) body is being expanded.

3) A question mark (?) or arrow (↑).

Respond to the prompt in any of the following ways:

<cr> Try to continue compilation. A message will be printed and the sequence reentered if recovery is impossible (if a "?" was typed instead of an arrow).

<lf> Try to continue the compilation, but don't stop for user response after future errors. I.e., automatic continuation. Messages will fly by (at an unreadable rate on DPYs) until the compilation is complete or an error occurs from which no recovery is possible. In the latter case the question sequence is reentered.

A same as <lf>

B Enter BAIL if it is loaded.

C same as <cr>

D Enter DDT or RAID if one is loaded. Otherwise, type "No DDT" and re-question. *Do not type D if you* really mean B.

E Edit. This command must be followed by a carriage return, or a space, a filename (in standard format, assumes DSK) and a carriage return. If the filename is missing, the SOS editor (see [Savitzky]) is started, given instructions to edit the current source file and to move the editing pointer to the current page and line number. If a file name is present, that file is edited starting at the beginning. This feature is available outside SUAI only if the SOS editor is available, and is modified to read a standard CCL file for its input. If you change your mind and do not wish to edit, typing an altmode will get you back to the question loop.

S Restart. Sometimes useful if you are debugging the compiler (or if you were compiling the wrong file). The program is restarted, accepting compilation commands from the TTY.

T TV edit. Same as E except that E is used at SUAI, TVEDIT at IMSSS and SUMEX.

X Exit. All files are closed in their current state. The program exits to the system.

Any other character will cause the error routines to spew forth a summary of this table and re-enter the question sequence.

ERROR MODES

For errors which occur during compilation, the above procedure can be modified slightly by setting various modes. One sets a mode by including the appropriate letter before the response. Any of the four modes may be reset by including a minus sign (-) before them. E.g.

"-Q". Error modes can also be set with REQUIRE <string_const> ERROR_MODES. When the compiler sees this it reads through the string constant and sets the modes as it sees their letters. These modes remain in effect until the end of the compilation or until reset with a response to an error message, or another require error_modes.

The available modes are:

K KEEP type-ahead. The error handler flushes all typeahead except a LF (linefeed). If KEEP mode is ever implemented then the input buffer will not be flushed.

L LOGGING. The first and second items of the error message will be sent to a file named <prognam>.LOG where <prognam> is the name of the file of the main program. If you would rather have another name, use F<file specification>, where <file specification> must be a legal file name and PPN. The default extension is .LOG and the default PPN is that of the job. The .LOG file (or whatever it's called) is closed when one's program finishes compilation, or the compilation is terminated with the S, X, E, or T responses.

N NUMBERS. This mode causes the message "Called from xxxx Last SAIL call at yyyy" to be typed before the question mark or arrow. Useful to compiler debuggers and hand coders.

Q QUIET. If the error is continuable, none of the above will be typed. However, you will always be notified of a non-continuable error.

Note that setting a mode does nothing but set a mode; it does not cause continuation.

### STOPPING RUNAWAY COMPILATIONS

Typing [ESC] I at SUAI or control-H on TENEX will immediately cause the Q and A modes to be reset so that the next error will (a) be typed, and (b) wait for a response rather than continuing automatically.

### EXECUTION TIME ERROR MESSAGES

Error messages have nearly the same format as those from the compiler (page 138). They indicate that

1) an array subscript has overflowed;

2) a case index is out of range;

3) a stack has overflowed while allocating space for a recursive procedure; or

4) one of the execution time routines has detected an error.

In Numbers mode, the "Called from" address identifies, in the first 3 cases, the location in the user program where the error occurred ; the "Last SAIL call at" address gives the location of the faulty call on the Sail routine for type 4 messages.

All the replies to error messages described in page 138 are valid. If no file name is typed with the "E" or "T" option, the editor re-opens the last file mentioned in the EDIT system command.

The function USERERR may be used to activate the Sail error message mechanism. Facilities are provided for changing the mode. See page 49 for details.

### USER ERROR PROCEDURES

A user error procedure is a user procedure that is run before or instead of the Sail error handler every time an error occurs at runtime. This includes all array errors, IO errors, Leapish errors and all USERERRs. It does not include system errors, such as III Mem Ref or III UUO.

The procedure one uses for a user error procedure must be of the following type:

```
SIMPLE INTEGER PROCEDURE proc
    (INTEGER loc; STRING msg, rsp);
```

Only the names proc, loc, msg, and rsp may vary from the example above, except that one may declare the procedure INTERNAL if one wishes to use it across files.

Whenever the external integer _ERRP_ is loaded with LOCATION (proc), the error handler

will call proc before it does anything else. It will set loc to the core location of the call to the error handler. Msg will be the message that it would have printed. Rsp will be non-NULL only if the error was from a USERERR which had response string argument. Proc can do anything that a simple procedure can do. When it exits, it should return an integer which tells the error handler if it should do anything more. If the integer is 0, the error handler will (1) print the message, (2) print the location, and (3) query the tty and dispatch on the response character (i.e., ask for a <cr>, <lf>, etc.). If the right half of the integer is non-zero, it is taken as the ascii for a character to dispatch upon. The left half may have two bits to control printing. If bit 17 in the integer is on, message printing is inhibited. If bit 16 is on, then the location printing is inhibited. For example, "X"+(1 LSH 18) will cause the location to be printed and the program exited. "C"+(3 LSH 18) will cause the error handler to continue without printing anything.

Note that simple procedures can not do a non-local GOTO. However, the effect of a non-local GOTO can be achieved in a user error procedure by loading the external integer _ERRJ_ with the LOCATION of a label. The label should be a on a call to a non-simple procedure which does the desired GOTO. The error handler clears _ERRJ_ before calling the procedure in _ERRP_. If _ERRJ_ is non-zero when the user procedure returns, and continuing was specified, then the error handler's exit consists of a simple transfer to that location. Note that for this simple transfer to work properly, the place where the error occurred (or the call to USERERR) must be in the same static (lexical) scope as the label whose LOCATION is in _ERRJ_. If this is really important to you, see a Sail hacker.

WARNING! Handling errors from strange places like the string garbage collector and the core management routines will get you into deep trouble.

## 23.2 Debugging

Sail has a high-level debugger called BAIL; see the description beginning in the next subsection. This subsection gives necessary information for those who wish to use DDT or RAID. The code output for Sail programs is designed to be fairly easy to understand when examined using the DDT debugging language or SUAI's display oriented RAID program. A knowledge of the debugger you have chosen is required before this section will be comprehensible.

### SYMBOLS

Only those symbols which have been declared INTERNAL (see page 12) and those declared in the currently open "program" are available at a given time. The name of a Sail program as far as DDT or RAID (henceforth DDRAID) is concerned is the name of the outer block of that program. If no name is given for this block, the name M. will be the default.

Only the first six non-blank characters of a block name or identifier will be used in forming a DDRAID symbol. If two identifiers in the same block have the same first six characters the program using them will not get confused, but the user might when trying to locate these identifiers.

### BLOCKS

All block names and identifiers used as variables, procedures or labels in a given (main or separate procedure) program are available for typeout when that program is "open" (NAMES: has been typed). To refer to a symbol, type BLOCK_NAME&SYMBOL/ (substitute ; for / in RAID). The block name may be omitted if you have "opened" the block with BLOCK_NAMES&. The symbol table is block-structured only to the extent that block names have appeared in the source program. For instance, in the program

```
BEGIN "NAME1"
  INTEGER I, J;
  ...
  BEGIN
    INTEGER I, K;
    ...
  END;
  ...
END "NAME1"
```

the symbols J, K, and both symbols I are considered by DDRAID to belong in the same block. Therefore confusion can result with respect to I. This approach was taken to avoid the necessity of generating meaningless block names for DDRAID when none were given in the source program. A compound statement will be considered by DDRAID to be a block if it has a name.

**SAIL GENERATED SYMBOLS**
Some extra symbols are generated by Sail and will show up when you are using DDRAID. They are:

ACS The accumulators P (system push down list pointer), and SP (string push down pointer) are given symbolic names. Currently P='17, SP='16.

OPS The op codes for the UUOs FIX, FLOAT, and ARERR (subscript overflow UUO) are included to make these easy to detect in the code.

ARRAYS *For each array declared* in the outer block (built-in arrays), the fixed address of its first element is given a symbolic name. This name is constructed from the characters of the array name (up to the first 5) followed by a period. For instance, the first element of array CHT is CHT.; the first element of PDQARR is PDQAR.; The last semicolon was *really a period.* This *dotted* symbol points to the second word of the first descriptor for String Arrays (see page 158, page 157).

STRINGS For each string declared in the outer block (built-in strings), the second word of the two word string descriptor is given the name of the string variable, truncated to six letters. The first word of the string descriptor is given a name consisting of the first five letters of the string's name followed by a period. For example, if you declare a string INSTRING, then the two word descriptor:

INSTR. : <first word>
INSTRI : <second word>

More about string descriptors on page 158.

BLOCKS The first word of the first executable statement of every block or compound statement which has been given a name is given a label created in the same way as those for arrays above. *This label cannot be gone to in* the source program. It causes no program inefficiency. This label points at the first word of the compound tail -- not the first word of code generated for the block (skips any procedure or array declaration code).

START The first word of code generated for any given program is given the name "S.".

PROCEDURES The word at entry address -1 of an INTERNAL procedure contains the address of the procedure descriptor. (This enables APPLY of an EXTERNAL procedure to work.) The first word of the procedure descriptor is given a name consisting of the first 5 characters of the procedure name, followed by a dollar sign ($).

**WARNINGS**
Since only the first 6 characters of an identifier are available, it is wise to declare symbols which will be examined by DDRAID in such a way that these six characters will uniquely identify them.

## 23.3 BAIL

BAIL [Reiser] is a high-level breakpoint package for use with Sail programs. Communication between the programmer and BAIL is in character strings which are the names and values of Sail objects. BAIL reads general Sail expressions typed by the programmer,

evaluates them in the context of the place in the program where execution was suspended, and prints the resulting value in an appropriate format. The evaluation and printing are performed just as if the programmer had inserted an extra statement into the original program at the point where execution was suspended. BAIL also provides a way to talk about the program, to answer the questions "Where was execution suspended?", "By what chain of procedure calls did execution proceed to that point?", and "What is the text of the program?"

In order to perform these functions, BAIL must have some information about the program being debugged. The Sail compiler will produce this information on a file with extension .SM1 if the program is compiled with an appropriate value supplied for the /B switch. The .SM1 information consists of the name, type, and accessing information for each variable and procedure, the location of the beginning and end of each statement, and a description of the block structure.

The code for BAIL itself is loaded automatically when the program is loaded. In order for the added information and code to be of any use, it must be possible to give control to BAIL at the appropriate time. An explicit call to BAIL is possible by declaring EXTERNAL PROCEDURE BAIL; in the program and using the procedure call BAIL;. This works well if it can be predicted in advance where BAILing might be helpful. Runtime errors, such as subscript overflow or CASE index errors, are not as predictable; but responding "B" to the Sail error handler will activate BAIL. Interrupting the program while it is running (to investigate a possible infinite loop, for example) can be achieved under the TENEX operating system by typing control-B. On a DEC TOPS-10 operating system, first return to monitor mode by typing one or more control-C's, then activate BAIL by typing DD<cr>.

BAIL performs some initialization the first time it is entered. The information in the .SM1 file(s) is collected and processed into a .BAI file. This new file reflects all of the information from the .SM1 files of any separately-compiled programs, and the relocation performed by the loader. If the core image was SAVEd or SSAVEd then in subsequent runs BAIL will use the .BAI file and bypass much of the initialization.

BAIL prompts the programmer for input by typing a number and a colon. The number indicates how many times BAIL has been entered but not yet exited, and thus is the recursion depth inside BAIL. Input to BAIL can be edited using the standard Sail input-editing characters for the particular operating system under which the program is running. [BAIL requests input via INCHWL on DEC TOPS-10 systems and via INTTY on TENEX systems.] Input is terminated whenever the editor activates, string quotation marks balance, and the last character is a semicolon; otherwise input lines are concatenated into one string before being processed further.

The programmer may ask BAIL to evaluate any Sail expression or procedure call whose evaluation would be legal at the point at which execution of the program being debugged was suspended (except that expressions involving AND, OR, IF-THEN-ELSE, and CASE are not allowed.) BAIL evaluates the expression, prints the resulting value in an appropriate format, and requests further input.

Declared inside BAIL are several procedures whose values or side effects are useful in the debugging process. These procedures handle the insertion and deletion of breakpoints, display the static and dynamic scope of the current breakpoint, display selected statements from the source program, allow escape to an assembly- language debugging program, and cause resumption of the suspended main program.

## COMPILE-TIME ACTION

The principal result of activating BAIL at compile-time is the generation of a file of information about the source program for use by the run-time interpreter. This file has the same name as the .REL file produced by the compilation, except that the extension is .SM1. If requested, BAIL will also generate some additional code for SIMPLE procedures to make them more palatable to the run-time interpreter.

The action of BAIL at compile time is governed by the value of the /B switch passed to the compiler. If the value of this switch is zero (the default if no value is specified) then BAIL is completely inactive. Otherwise, the low-order bits determine the actions which BAIL performs. [The value of the /B switch is interpreted as octal.]

| bit | action if on |
|---|---|
| 1 | The .SM1 file will contain the program counter to source/listing text directory. |
| 2 | The .SM1 file will contain symbol information for all Sail symbols encountered in the source. If this bit is off, then information is kept only for procedures, parameters, blocks, and internals; i.e., non-internal local variables are not recorded. |
| 4 | SIMPLE procedures will get procedure descriptors, and one additional instruction (a JFCL 0) is inserted at the beginning of SIMPLE procedures. Except for these two changes, all properties of SIMPLE procedures remain the same as before. The procedure descriptor is necessary if the procedure is to be called interpretively or if the procedure is to be TRACEd. |
| '10 | BAIL will not be automatically loaded and initialized, although all other actions requested are performed. This is primarily intended to make it easier to debug new versions of BAIL without interfering with SYS:BAIL.REL. By using this switch the decision to load BAIL is delayed until load time. |
| '20 | A request to load SYS:BAIPDn.REL is generated. This file contains requests to load procedure descriptors for most of the predeclared runtime routines, making it possible to call them from BAIL. The procedure descriptors and their symbols occupy about 12P. Subsets of these procedure descriptors can be loaded individually to reduce memory space requirements, at the cost of not being able to talk about the routines omitted. The subsets are BAICLC (containing SQRT, EXP, LOG, SIN, COS, RAN, CVOS, CVSTR, CVXSTR), BAII01 (major input/output and string procedures), BAII02 (minor input/output and string procedures), BAIMSC (terminal functions and miscellaneous), and BAIPRC (process and interrupt routines). To use these subsets, request them explicitly (e.g., REQUIRE "SYS:BAICLC" LOAD_MODULE; or on TENEX, "<SAIL>BAICLC") and leave the /20B bit off. |

The B switch must occur on the binary term, not the listing or source term. Thus:

```
.R SAIL          or      .COM PROG(27B,)
*PROG/27B←PROG
```

The program counter to source/listing index is kept in terms of coordinates. The coordinate counter is zeroed at the beginning of the compilation and is incremented by one for each BEGIN, ELSE, and semicolon seen by the parser, provided at least one word of code has been compiled since the previous coordinate was defined. Note that COMMENTs are seen only by the scanner, not the parser, and that DEFINEs and many declarations merely define symbols and do not cause instructions to be generated. For each coordinate the directory contains the coordinate number, the value of the program counter, and a file pointer to the appropriate place. The appropriate place is the source file unless a listing file is being produced and the CREF switch is off, in which case it is the listing file. [The listing file produced for CREF is nearly unreadable.] On a non-CREF listing, the program counter is replaced by the coordinate number if bit 1 of the /B switch is on.

The symbol table information consists of the block structure and the name, access information, and type for each symbol.

If a BEGIN-END pair has declarations (i.e., is a true block and not just a compound statement) but does not have a name, then BAIL will invent one. The name is of the form Bnnnn where nnnn is the decimal value of the current coordinate.

### RUN-TIME ACTION

The BAIL run-time interpreter is itself a Sail program which resides on the system disk area. This program is usually loaded automatically, and does some initialization when entered for the first time. The initialization generates a .BAI file of information collected from the .SM1 files produced by separate compilations (if any). The .SM1 files correspond to .REL files, and the .BAI file corresponds to the .DMP or .SAV file. Like RPG or CCL, BAIL will try to bypass much of the initialization and use an existing .BAI file if appropriate. During initialization BAIL displays the names of the .SM1 files it is processing. For each .SM1 file which

contains program counter/text index information, BAIL displays the names of the text files and determines whether the text files are accessible.

The interpreter is activated by explicit call, previously inserted breakpoints, or the Sail error handler. For an explicit call, say EXTERNAL PROCEDURE BAIL; ... BAIL;. From the error handler, respond B. Breakpoints will be described later in this section.

### DEBUGGING REQUESTS

When entered, BAIL prints the debugging recursion level followed by a colon, and awaits a debugging request. BAIL accepts ALGOL and LEAP expressions of the Sail language. The following exceptions should be noted. Expressions involving control structure are not allowed, hence BAIL will not recognize AND, OR, IF-THEN-ELSE, or CASE. Bracketed triple items are not allowed. The TO and FOR substring and sublist operators have been extended to operate as array subscript ranges, FOR PRINT-OUT ONLY. If FOO is an array, then FOO[3 TO 7]; will act like FOO[3], FOO[4], FOO[5], FOO[6], FOO[7]; but is easier to type. This extension is for print-out only; no general APL syntax or semantics are provided.

BAIL evaluates symbolic names according to the scope rules of ALGOL, extended to always recognize names which are globally unique and have a fixed memory location (everything except parameters and recursive locals). For any activation of BAIL, the initial scope is the ALGOL scope of the statement from which BAIL was activated. The procedure SETLEX (see below) may be used to change the scope to that of any one of the links in the dynamic activation chain. See also the section below on BLOCK STRUCTURE for a way to evade the scope rules.

Several procedures are predeclared in the outermost block to handle breakpoints and display information. These are described individually below.

#### ARGS

"STR" ← ARGS

The arguments to the procedure which was most recently called.

#### BREAK

BREAK ("LOCATION", "CONDITION"(NULL),
"ACTION"(NULL), COUNT(0))

A breakpoint is inserted. The syntax for the first argument is

<location>
::= <label>
::= <procedure>
::= <block name>
::= #<nnnn>
::= <block name> . <location>

<nnnn>
::= <decimal coordinate number>

If the location is specified by the <block name>.<location> construct then the blocks of the core image are searched in ascending order of address of BEGINs until the first <block name> is matched. The search continues until the second <block name> is matched, etc. The breakpoint is inserted at the label, procedure, or coordinate declared within the scope of the last <block name>. This detailed specification is not usually necessary. The action taken at a breakpoint is

```
IF LENGTH (CONDITION) AND EVAL (CONDITION)
  AND (COUNT ← COUNT-1)<0 AND LENGTH(ACTION)
THEN EVAL(ACTION);
EVAL(TTY)
```

#### COORD

NUMBER ← COORD ("LOCATION")

Returns the coordinate number of the location given as its argument. LOCATION has the same syntax as in BREAK.

### DDT

DDT

This procedure transfers control to an assembly language debugging program (if one was loaded).

### DEFINE

DEFINE (CHAR, "MACRO")

Macros from the source file(s) are not recognized at the present time. There are 26 character macros definable, from "A" to "Z". DEFINE macros substitute the given string for each occurrence of <alt><char> which is not part of a string constant. If the operating system can send characters of more than 7 bits to INCHWL (INTTY under TENEX) then any activation character with high order bits will also activate the macro. Thus at SUAI <alt>P, $\alpha$P, and $\alpha\beta$P are all equivalent. In all cases the character is converted to upper case before doing anything else. The macros G, P, S, and X are predefined to be " !!GO;", " !!GO;", " !!STEP;", and " !!GSTEP;" respectively.

### HELP

HELP

A list of options, including short descriptions of the procedures described in this section, is printed. An input consisting of a question mark followed by a carriage return is interpreted as a call to HELP.

### SETLEX

SETLEX (LEVEL)

Evaluating SETLEX(n) changes the static (lexical) scope to the scope of the n-th entry in the dynamic scope list. SETLEX(0) is the scope of the breakpoint; SETLEX(1) is the scope of the most recent procedure call in the dynamic scope, etc.

### SHOW

"STR" ← SHOW (FIRST, LAST(0))

The text of the program from the source or listing file. If last is less than first then set last to last+first. Return coordinates first through last. SHOW (5, 3) gives coordinates 5, 6, 7, and 8; SHOW (5, 7) gives coordinates 5, 6, and 7; SHOW (5) gives coordinate 5 only.

A plus sign ("+") following the coordinate number indicates that the values of some variables have been carried over in accumulators from the previous coordinate. Changing the value of variables might not be successful in such a case, because BAIL will not change any accumulator value directly. The MEMORY construct can be used to modify any location in a core image, including the accumulators.

### TEXT

"STR" ← TEXT

The current static and dynamic scopes, with text from the source or listing file.

### TRACE

TRACE ("PROCEDURE")

Special breakpoints are inserted at the beginning and end of the procedure named. On entry, the procedure name and arguments are typed. On exit, the name and value returned (if any) are typed.

### TRAPS

"STR" ← TRAPS

A list of the current breakpoints and traces.

### UNBREAK

UNBREAK ("LOCATION")

The breakpoint at the location specified is removed.

### UNTRACE

UNTRACE ("PROCEDURE")

The breakpoints inserted by TRACE are removed.

### !!GO

!!GO

An immediate exit from the current instantiation of BAIL is taken and execution of the program is resumed. !!GO is a reserved word (the only one) in BAIL.

### !!GSTEP

!!GSTEP

Temporary breakpoints are inserted at all of the logical exits of the current statement, and execution of the program is resumed. Logical exits are the next statement and locations to which the current statement can jump, excluding any procedure calls. All of the breakpoints which are inserted will be removed as soon as one of them is encountered.

### !!STEP

!!STEP

Temporary breakpoints are inserted at all locations to which the current statement can jump, including procedure calls, and execution of the program is resumed.

### GOGTAB

EXTERNAL INTEGER ARRAY GOGTAB[0:n]

This array is the Sail user table, containing all kinds of magical information. (The procedure USERCON was formerly the only way to access the user table.) If you are a hacker then pick up a copy of SYS:GOGTAB.DEF (<SAIL>GOGTAB.DEF on TENEX) and poke around. Do not change any values unless you know what you are doing.

STRING TYPEOUT
Strings are usually typed so that the output looks the same as the input, i.e., a string is typed with surrounding quotation marks and doubled internal quotation marks. For SHOW, ARGS, and TEXT this would ordinarily create confusion, so they are handled specially. When these procedures are evaluated they set a flag which inhibits quotation mark fiddling, provided that no further evaluation takes place before the next typeout. Thus SHOW (5, 3); will be typed plain, but STR ← SHOW (5, 3); will have quotation marks massaged.

BLOCK STRUCTURE
Variables not in the current scope can be referenced by using the same scheme used to describe locations to BREAK. If you have something of your own named SHOW then you can access the BAIL SHOW function by using $RUNS.SHOW (coord);. Warning: this mode assumes that you know what you are doing.

BAIL and DDT
When BAIL is loaded by a non-TENEX system, it sets .JBDDT to the address of one of its routines. (If you load both BAIL and DDT then the last module loaded wins.) Under TENEX, BAIL sets .JBDDT at runtime, but only if it is zero when BAIL looks. If BAIL is initialized in a core image which does not have DDT or RAID then things will be set up so that the monitor command DDT gets you into BAIL in the right way. That is, BAIL will be your DDT. To enter BAIL from DDT (provided that the Sail initialization sequence has already been performed), use

```
pushj P,<program counter>$X
JRST BAIL$X
```

For example, if .JBOPC contains the program counter,

```
PUSH P,.JBOPC$X
JRST BAIL$X
```

The entry B. provides a path from DDT to BAIL which works whether or not the core image has been initialized. One use of this feature is to BREAK a procedure in an existing production program without recompiling. For example,

```
@; PROG compiled, loaded with BAIL and DDT, and SSAVEd
@GET PROG
@DD
 B$G
BAIL initialization
        :
        :
1:BREAK("procedure");
1:!!GO;

$G
```

To enter DDT from BAIL, simply say DDT;. For operation under TENEX, control-B is a pseudo-interrupt character which gets you into BAIL.

WARNINGS
Since BAIL is itself a Sail procedure, entering BAIL from the error handler or DDT after a push-down overflow or a string garbage collection error will get you into trouble.

SIMPLE procedures cause headaches for BAIL because they do not keep a display pointer. BAIL tries to do the right thing, but occasionally it gets lost. BAIL will try to warn you if it can. In general, looking at value string parameters of SIMPLE procedures does not work.

!!GOTO ("LOCATION")
(For wizards only.) The return address is set to the location specified, and then a !!GO is done. Note that the location should be in the same lexical scope as the most recent entry to BAIL, or the program will probably get confused.

!!UP (LEVEL)
(For wizards only.) This procedure trims the runtime stack back to LEVEL, then reenters BAIL. CLEANUPs and deallocations are performed for the procedures thus killed. Level has the same interpretation as in SETLEX, and in addition must not designate a SIMPLE procedure. Suppose you ask BAIL to evaluate a procedure call, the procedure hits an error, and you want to get back to where you were before the procedure was called. Then !!UP will do the trick if the value of level is correct.

!!QUERY
(Declare as EXTERNAL STRING !!QUERY in your program.) Whenever BAIL wants input, it checks this string first. If it is not NULL then !!QUERY is used instead of asking the operating system for input from the terminal. (!!QUERY is set to NULL each time this is done.) Thus a program can simulate the effect of typing to its own input buffer by stuffing the text into !!QUERY. In particular, file input to BAIL and various macro hacks can be effected by using procedures which assign values to !!QUERY.

## SETSCOPE

SETSCOPE (ITEMVAR PITEM)

If you have processes then SETSCOPE can be used to peek around the world. Specifically, the static and dynamic scopes are set to those of the process for which PITEM is the process item. This will allow access to variables and traceback from TEXT, but care must be exercised when calling procedures. A call to a procedure which is not defined at the top level will probably not work. Also, if the procedure does not return successfully then your stacks will be hopelessly confused.

Note on processes: BAIL runs in the process which caused the break. Thus stack space must be provided in each process. The minimum amount is PSTACK(4)+STRINGSTACK(2).

RESOURCES USED
At compile time one channel, a small amount of additional memory, and approximately 0.3 seconds of KA10 CPU time per page are used. BAIL uses two channels at runtime and a third during initialization. These channels are obtained with GETCHAN. If the debugging recursion level exceeds 3 or 4 then it will be necessary to increase the pushdown stacks (particularly STRING_PDL) appropriately. BAIL uses 7 of the privileged breaktables, obtaining them with GETBREAK. BAIL occupies 19.5 pages. Symbols require 5 words each with an additional 2 words for each block; one word for each 128 coordinates is also required. The disk space required for .SWI and .BAI files is

generally one half that required for the .REL files.

## EXAMPLE

```
@TYPE TEST.SAI

, <REISER>TEST.SAI.1   SAT 28-AUG-76 4:20PM    PAGE 1


BEGIN "TEST"
EXTERNAL PROCEDURE BAIL;
INTEGER I, J, K; STRING A, B, C; REAL X, Y, Z;
INTEGER ARRAY FOO[0:15]; STRING ARRAY STRARR[1:5, 2:6];

INTEGER PROCEDURE ADD (INTEGER I, J); BEGIN "ADD"
OUTSTR ("
Hi. GLAD YOU STOPPED BY."); RETURN (I+J) END "ADD";

FOR I←0 STEP 1 UNTIL 15 DO FOO[I]←I+I;
FOR I←1 STEP 1 UNTIL 5 DO
    FOR J←2 STEP 1 UNTIL 6 DO STRARR[I, J]←64+8*I+J;
I←4; J←6; K←112; X←3.14159265; Y←0; Z←23;
A←"BIG DEAL"; B←"QED"; C←"THE LAST PICASSO";

BAIL; ADD (7, 45); USERERR (0, 1, "THIS IS A TEST");
END "TEST";
↑L

@SAIL SAV;10
 TENEX SAIL 8.1 8-28-76  (? FOR HELP)
*TEST.1,←
**/27B
**
TEST.SAI.1  1
END OF COMPILATION.
LOADING

LOADER 6+9K CORE
EXECUTION

$G
BAIL VER 28-AUG-76
TEST.SM1;2
  TEST.SAI;1
End of BAIL initialization.

1:45, 7089, "SOME RANDOM STRING";
   45    7089000    "SOME RANDOM STRING"
1:'275, TRUE, FALSE, NULL;
   189   -1   0   ""
1:J, X, I←46;
   6    3.141593      46
1:I, I<J;
   46   0
1:45*(894-5306);
    1635300
1:ADD (3, 4);

Hi. GLAD YOU STOPPED BY.   7
1:FOO;
   <ARRAY>[ 0:15]
1:FOO[4];
   16
1:STRARR[1 FOR 2, 4 TO 6];
   "L"   "M"   "N"   "T"   "U"   "V"

1:FOO[35];

SUBSCRIPTING ERROR
  INDEX   VALUE   MIN   MAX
    1       35     0     15     : FOO[35]
                                    ;

1:BREAK ("ADD");

1:ADD (3, 4);

2:ARGS;
  3   4
2:!!GO;

Hi. GLAD YOU STOPPED BY.   7
1:!!GO;

1:TEXT;
LEXICAL SCOPE, TOP DOWN:
$RUN$
TEST
ADD

ROUTINE        TEXT
ADD    *4      INTEGER PROCEDURE ADD (INTEGER I, J);
TEST   *24     ADD (7, 45);

AT SETLEX(0);

1:UNBREAK ("ADD");

1:!!GO;
Hi. GLAD YOU STOPPED BY.
THIS IS A TEST
CALLED FROM 642124  LAST SAIL CALL AT 400303
↑B
1:TEXT;

LEXICAL SCOPE, TOP DOWN:
$RUN$

DYNAMIC SCOPE, MOST RECENT FIRST:
ROUTINE        TEXT
.SIMPLE.       '642124 ??? FILE NOT VIEWABLE
TEST   *26     USERERR (0, 1, "THIS IS A TEST");

AT SETLEX(0);
1:!;

UNKNOWN ID:  I
               ;
1:SETLEX (1);

LEXICAL SCOPE, TOP DOWN:
$RUN$
TEST

1:!;
  64
1:!!GO;

END OF SAIL EXECUTION.
```

## CURRENT STATUS

The state of the world is determined by the values of the accumulators and the value of the Sail variable _SKIP_.

The run-time interpreter recognizes only the first 15 characters of identifier names; the rest are discarded without comment. The characters which are legal in identifiers are

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789!_αβnλc⊃∀3¬~#$\|

Notable for its absence: period.

LOCATION of a procedure does not work.

PROPS is read-only.

Bracketed triple items are not allowed.

A procedure call containing the name of a parametric procedure (functional argument) is not handled properly.

Contexts are not recognized.

External linkage: If an identifier is never referenced by code (i.e., has an empty fixup chain at the time fixups are put out to the loader) then that identifier is not defined by Sail. Thus variables which are never used do not take up space and a request to the loader is not made for EXTERNALS which are not referenced. This feature of Sail. As a result, the following DOES NOT WORK unless special precautions are taken:

```
BEGIN
EXTERNAL PROCEDURE BAIL;
EXTERNAL PROCEDURE
   PLOT (REAL X0, Y0, X1, Y1);
REQUIRE "CALCOM" LIBRARY;

BAIL END
```

PLOT will not be defined by Sail, hence BAIL will not know about it. However if there are any references to PLOT (real or "dummy" calls) then BAIL will know. The following trick can also be used, assuming that CALCOM is a Sail-compiled library: Compile CALCOM with /10B, which says "make the .SM1 file but don't automatically load SYS:BAIL.REL". Then the above will win (due to BAIL recognizing things which are globally unique) and programs which do not use BAIL will not have it loaded just because the library was used. This same problem occurs with EXTERNAL RECORD_CLASS declarations. Use of the field index information does not cause a reference to the class name but NEW_RECORD does. Thus the same /10B trick must be used if there are no NEW_RECORD calls.

BAIL and other language processors: If CALCOM in the paragraph above was compiled by some processor other than Sail (e.g. FAIL, MACRO, BLISS, ...) then further steps must be taken if BAIL is to know about the procedures contained in the file. BAIL must have access to a procedure descriptor in order to call any procedure (cf. the /4B switch). Thus a user who wishes to use assembly language procedures with BAIL must provide appropriate procedure descriptors. The file ⊂SUAI⊃SAILPD.FAI[S,AIL] defines a FAIL macro which will generate a Sail procedure descriptor. The procedure descriptors may reside in a separate load module if desired; but they must be in the core image when BAIL is being used.

# APPENDIX A

## Characters

| CHARACTER | EQUIVALENT RESERVED WORD |
|---|---|
| ∧ | AND |
| ≡ | EQV |
| ¬ | NOT |
| ∨ | OR |
| ⊗ | XOR |
| ∞ | INF |
| ∈ | IN |
| \| | SUCH THAT |
| ≠ | NEQ |
| ≤ | LEQ |
| ≥ | GEQ |
| { | SETO |
| } | SETC |
| ∪ | UNION |
| ∩ | INTER |
| ' | ASSOC |
| ↔ | SWAP |
| _ | ! |

**Stanford (SUAI) Character Set**

The Stanford ASCII character set is displayed in the following table. The three digit octal code for a character is composed of the number at the left of its row plus the digit at the top of its column. For example, the code for "A" is 100+1 or 101.

| | ASCII | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | ↓↓↓ | | | | | | | | |
| | 000 | NUL | ↓ | α | β | ∧ | ¬ | ∈ | π |
| | 010 | λ | TAB | LF | VT | FF | CR | ∞ | ∂ |
| SIXBIT | 020 | ⊂ | ⊃ | ∩ | ∪ | ∀ | ∃ | ⊗ | ↔ |
| ↓↓ | 030 | _ | → | ~ | ≠ | ≤ | ≥ | ≡ | ∨ |
| 00 | 040 | SP | ! | " | # | $ | % | & | ' |
| 10 | 050 | ( | ) | * | + | , | - | . | / |
| 20 | 060 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 30 | 070 | 8 | 9 | : | ; | < | = | > | ? |
| 40 | 100 | @ | A | B | C | D | E | F | G |
| 50 | 110 | H | I | J | K | L | M | N | O |
| 60 | 120 | P | Q | R | S | T | U | V | W |
| 70 | 130 | X | Y | Z | [ | \ | ] | ↑ | ← |
| | 140 | ' | a | b | c | d | e | f | g |
| | 150 | h | i | j | k | l | m | n | o |
| | 160 | p | q | r | s | t | u | v | w |
| | 170 | x | y | z | { | \| | ALT | } | BS |

The tables below display the standard ASCII codes, and the SOS representation for entering the full ASCII character set from Teletypes or similar terminals with restricted character sets. The obscure names for the ASCII codes below 40 are listed just for confusion. Notes: "DEL" (177) is the ASCII delete. "ESC" (33) is their alt mode. Codes 136 and 137 have two different interpretations, as shown below. The SOS representation is so called because it is provided by SOS, the Teletype editor. Certain other programs also know about this representation, but it is not built into Sail in any way.

**Standard ASCII**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 000 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL |
| 010 | BS | TAB | LF | VT | FF | CR | SO | SI |
| 020 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB |
| 030 | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 040 | SP | ! | " | # | $ | % | & | ' |
| 050 | ( | ) | * | + | , | - | . | / |
| 060 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 070 | 8 | 9 | : | ; | < | = | > | ? |
| 100 | @ | A | B | C | D | E | F | G |
| 110 | H | I | J | K | L | M | N | O |
| 120 | P | Q | R | S | T | U | V | W |
| 130 | X | Y | Z | [ | \ | ] | ^↑ | _← |
| 140 | ` | a | b | c | d | e | f | g |
| 150 | h | i | j | k | l | m | n | o |
| 160 | p | q | r | s | t | u | v | w |
| 170 | x | y | z | { | \| | } | ~ | DEL |

**SOS Representation of Standard ASCII**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 000 | --- | ?! | ?" | ?# | ?$ | ?% | ?& | ?' |
| 010 | ?( | TAB | LF | VT | FF | CR | ?) | ?* |
| 020 | ?+ | ?, | ?- | ?. | ?/ | ?0 | ?1 | ?2 |
| 030 | ?9 | ?6 | ?4 | ?= | ?< | ?> | ?7 | ?8 |
| 040 | SP | ! | " | # | $ | % | & | ' |
| 050 | ( | ) | * | + | , | - | . | / |
| 060 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 070 | 8 | 9 | : | ; | < | = | > | ?? |
| 100 | @ | A | B | C | D | E | F | G |
| 110 | H | I | J | K | L | M | N | O |
| 120 | P | Q | R | S | T | U | V | W |
| 130 | X | Y | Z | [ | \ | ] | ↑ | ← |
| 140 | ?@ | ?A | ?B | ?C | ?D | ?E | ?F | ?G |
| 150 | ?H | ?I | ?J | ?K | ?L | ?M | ?N | ?O |
| 160 | ?P | ?Q | ?R | ?S | ?T | ?U | ?V | ?W |
| 170 | ?X | ?Y | ?Z | ?[ | ?\ | ?] | ?3 | ?\ |

The Sail compiler automatically transliterates "!" to "_" before doing anything else (outside of string constants, of course). It also believes that BOTH '175 and '176 represent the right brace character "}".

# APPENDIX B

## Sail Reserved Words

ABS
ACCESS
AFTER
ALL
ALLGLOBAL
AND
ANY_CLASS
APPLY
ARG_LIST
ARRAY
ASH
ASSIGN
ASSIGNC
ASSOC
BBPP
BEFORE
BEGIN
BIND
BOOLEAN
BUCKETS
BUILT_IN
CASE
CASEC
CAUSE
CHECK_TYPE
CLEANUP
COMMENT
COMPILER_SWITCHES
CONOK
CONTEXT
CONTINUE
COP
CPRINT
CVI
CVLIST
CVMS
CVN
CVPS
CVSET
DATUM
DECLARATION
DEFINE
DELETE
DELIMITERS
DEPENDENTS
DIV
DO
DOC
DONE
DPB
ELSE
ELSEC
END
ENDC
ENTRY
EQV
ERASE
ERROR_MODES
EVALDEFINE
EVALREDEFINE
EXPR_TYPE
EXTERNAL
FAIL
FALSE
FIRST
FOR
FORC
FOREACH
FORGET
FORLC
FORTRAN
FORWARD
FROM
GEQ
GLOBAL
GO
GOTO
IBP
IDPB
IF
IFC
IFCR
ILDB
IN
IN_CONTEXT
INF
INITIALIZATION
INTEGER
INTER
INTERNAL
INTERROGATE
ISTRIPLE
ITEM
ITEM_START
ITEMVAR
KILL_SET
LABEL
LAND
LDB
LEAP_ARRAY
LENGTH
LEQ
LET
LIBRARY
LIST
LISTC
LISTO
LNOT
LOAD_MODULE
LOCATION
LOP
LOR
LSH
MAKE
MATCHING
MAX
MEMORY
MESSAGE
MIN
MOD
NEEDNEXT
NEQ
NEW
NEW_ITEMS
NEW_RECORD
NEXT
NIL
NOMAC
NOT
NOW_SAFE
NOW_UNSAFE
NULL
NULL_CONTEXT
NULL_DELIMITERS
NULL_RECORD
OF
OFC
OR
OWN
PHI
PNAMES
POLL
POLLING_INTERVAL
PRELOAD_WITH
PRESET_WITH
PRINT
PROCEDURE
PROCESSES
PROTECT_ACS
PUT
QUICK_CODE
REAL
RECORD_CLASS
RECORD_POINTER
RECURSIVE
REDEFINE
REF_ITEM
REFERENCE
REMEMBER
REMOVE
REPLACE_DELIMITERS
REQUIRE
RESERVED
RESTORE
RETURN
ROT
SAFE
SAMEIV
SECOND
SEGMENT_FILE
SEGMENT_NAME
SET
SETC
SETCP
SETIP
SETO
SHORT
SIMPLE
SOURCE_FILE
SPROUT
SPROUT_DEFAULTS
START_CODE
STEP
STEPC
STRING
STRING_PDL
STRING_SPACE
SUCCEED
SUCH
SWAP
SYSTEM_PDL
THAT
THEN
THENC
THIRD
TO
TRUE
UNION
UNSTACK_DELIMITERS
UNTIL
UNTILC
VALUE
VERSION
WHILE
WHILEC
XOR

# APPENDIX C

## Sail Predeclared Identifiers

$PINT
$PITM
$PLST
$PREC
$PREL
$PRINT
$PSET
$PSTR
ACOS
ANSWER
ARRBLT
ARRCLR
ARRINFO
ARRTRAN
ARRYIN
ARRYOUT
ASIN
ASKNTC
ATAN
ATAN2
BBPP.
BINDIT
BREAKSET
CALL
CALLER
CAUSE1
CHNCDB
CLKMOD
CLOSE
CLOSIN
CLOSO
CLRBUF
CODE
COMPILER_BANNER
COS
COSD
COSH
CV6STR
CVASC
CVASTR
CVD
CVE
CVF
CVFIL
CVG
CVIS
CVO
CVOS
CVS
CVSI
CVSIX
CVSTR
CVXSTR
DDFINT
DEL_PNAME
DFCPKT
DFRIIN
DFRINT
DISABLE
EOFILE
ENABLE
ENTER
EQU
ERMSBF
EVENT_TYPE
EXP
FILEINFO
GETBREAK
GETCHAN
GETFORMAT
GETPRINT
INCHRS
INCHRW
INCHSL
INCHWL
INPUT
INSTR
INSTRL
INSTRS
INTIN
INTMAP
INTPRO
INTSCAN
INTSET
INTTBL
JOIN
LINOUT
LISTX
LOG
LOOKUP
MAINPI
MAINPR
MKEVTT
MTAPE
MYPROC
NEW_PNAME
OPEN
OUT
OUTCHR
OUTSTR
POINT
PRISET
PSTATUS
RAN
REALIN
REALSCAN
RELEASE
RENAME
RESUME
SCAN
SCANC
SETBREAK
SETFORMAT
SETPL
SETPRINT
SIN
SIND
SINH
SQRT
STDBRK
SUBSR
SUBST
SUSPEND
TANH
TERMINATE
TRIGINI
TTYIN
TTYINL
TTYINS
TTYUP
TYPEIT
URSCHD
USERCON
USERERR
USETI
USETO
WORDIN
WORDOUT

**SUAI ONLY**

GET_BIT
GET_DATA
GET_ENTRY
GET_SET
IFGLOBAL
ISSUE
LODED
PTCHRS
PTCHRW
PTIFRE
PTOCHS
PTOCHW
PTOCNT
PTOSTR
PTYALL
PTYGET
PTYIN
PTYREL
PTYSTR
PUT_DATA
QUEUE

**TOPS-10 ONLY**

BACKUP
CHNCDB
ERENAME
INOUT
GETSTS
SETSTS
INTMOD
TMPIN
TMPOUT

**CMU ONLY**

ARDINIT
ARDSTR
CHARSZ
CHRMOD
CLEAR
DOTVEC
INITSEA
INVVEC
MOUSES
MOUSEW
SEAINIT
SEAREL
SETPNT
SVEC
VISVEC

**TYMSHARE ONLY**

AUXCLR
AUXCLV
CALLI
CHNIOR
CHNIOV
IONEOU

**TENEX ONLY**

ASND
ATI
BKJFN
CFILE
CHARIN
CHAROUT
CHFDB
CLOSF
CNDIR
CVJFN
DEVST
DEVTYPE
DIRST
DTI
DVCHR
ERSTR
GDSTS
GJINF
GNJFN
GTAD
GTFDB
GTJFN
GTRPW
GTSTS
IDTIM
INDEXFILE
INTTY
JFNS
JFNSL
KPSITIME
MTOPR
ODTIM
OPENF
OPENFILE
PBIN
PBOUT
PMAP
PSIDISMS
PSIMAP
PSIRUNTM
PSOUT
RCHPTR
RDSEG
RELD
RFBSZ
RFCOC
RFMOD
RFPTR
RLJFN
RNAMF
RTIW
RUNPRG
RUNTM
RWDPTR
SCHPTR
SDSTS
SETCHAN
SETEDIT
SETINPUT
SFCOC
SFMOD
SFPTR
SINI
SIZEF
STDEF
STDIR
STI
STIW
STPAR
STSTS
STTYP
SWDPTR
UNDELETE

# APPENDIX D

## Indices for Interrupts

### SUAI INTERRUPT SYSTEM

| NAME | NUMBER | DESCRIPTION |
|---|---|---|
| INTSWW_INX | 0 | You will receive an interrupt when your job is about to be swapped out. |
| INTSWD_INX | 1 | You will receive an interrupt when your job is swapped back into core. If you are activated for interrupts for swap out also, you will receive these two interrupts as a pair in the expected order every time your job is swapped. |
| INTSHW_INX | 2 | You will receive an interrupt when your job is about to be shuffled. |
| INTSHD_INX | 3 | You will receive an interrupt when your job has been shuffled. |
| INTTTY_INX | 4 | You will receive an interrupt every time your program would be activated due to the teletype if it were waiting for the teletype. As long as you do not ask for more than there is in the teletype buffer, you may read from the teletype at interrupt level. |
| INTPTO_INX | 5 | You will be interrupted every time the PTY job goes into a wait state waiting for you to sent it characters. |
| INTMAIL_INX | 6 | Interrupts whenever someone SENDs you mail (see [Frost]). You may read the letter at interrupt level. |
| INTPTI_INX | 8 | You will be interrupted every time any job on a PTY you own send you a character (or line). |
| INTPAR_INX | 9 | Interrupts you on parity errors in your core image. |
| INTCLK_INX | 10 | You will be interrupted at every clock tick (1/60th of a second). |
| INTINR_INX | 11 | IMP interrupt by receiver. |
| INTINS_INX | 12 | IMP interrupt by sender. |
| INTIMS_INX | 13 | IMP status change interrupt. |
| INTINP_INX | 14 | IMP input waiting. |
| INTTTI_INX | 15 | You will be interrupted whenever <esc> I is typed on your teletype. |
| INTPOV_INX | 19 | Interrupts you on push-down overflow. |
| INTILM_INX | 22 | Interrupts you on illegal memory references, that is, references to memory outside of your core image. |
| INTNXM_INX | 23 | You will receive an interrupt whenever your program references non-existent memory. |
| INTFOV_INX | 29 | Interrupts you on floating overflow. |
| INTOV_INX | 32 | Interrupts you on arithmetic overflow. |

Bits 33 through 35 are left to the user. REQUIRE "SYS:PROCES.DEF" SOURCE_FILE to define the above names. NOTE: to program yourself for more than one interrupt, you must execute two separate INTMAP statements.

## TOPS-10 INTERRUPT SYSTEM

| NAME | NUMBER | DESCRIPTION |
|---|---|---|
| INTPOV_APR | 19 | Interrupts you on push-down stack overflow. |
| INTILM_APR | 22 | Interrupts you on illegal memory references, that is, references to memory outside of your core image. |
| INTNXM_APR | 23 | You will receive an interrupt whenever your program references non-existent memory. |
| INTFOV_APR | 29 | Interrupts you on floating overflow. |
| INTOV_APR | 32 | Interrupts you on arithmetic overflow. |

## TENEX PSI CHANNELS

| CHANNEL | USE |
|---|---|
| 0-5 | terminal character |
| 6 | APR integer overflow, no divide |
| 7 | APR floating overflow, exponent underflow |
| 8 | unused |
| 9 | pushdown overflow |
| 10 | file EOF |
| 11 | file data error |
| 12 | file, unassigned |
| 13 | file, unassigned |
| 14 | time of day |
| 15 | illegal instruction |
| 16 | illegal memory read |
| 17 | illegal memory write |
| 18 | illegal memory execute |
| 19 | subsidiary fork termination, forced freeze |
| 20 | machine size exceeded |
| 21 | SPACS trap to user |
| 22 | reference to non-existent page |
| 23 | unused |
| 25-35 | terminal character |

# APPENDIX E

## Bit Names for Process Constructs

SPROUT OPTIONS

| BITS | NAME | DESCRIPTION |
|---|---|---|
| 14-17 | QUANTUM(X) | Q ← IF X=0 THEN 4 ELSE 2↑X; The process will be given a quantum of Q clock ticks, indicating that if the user is using CLKMOD to handle clock interrupts, the process should be run for at most Q clock ticks, before calling the scheduler. (see about CLKMOD, page 120 for details on making processes "time share"). |
| 18-21 | STRINGSTACK(X) | S ← IF X=0 THEN 16 ELSE X*32; The process will be given S words of string stack. |
| 22-27 | PSTACK(X) | P←IF X=0 THEN 32 ELSE X*32; The process will be given P words of arithmetic stack. |
| 28-31 | PRIORITY(X) | P ← IF X=0 THEN 7 ELSE X; The process will be given a priority of P. 0 is the highest priority, and reserved for the Sail system. 15 is the lowest priority. Priorities determine which ready process the scheduler will next pick to make running. |
| 32 | SUSPHIM | If set, suspend the newly sprouted process. |
| 33 | | Not used at present. |
| 34 | SUSPME | If set, suspend the process in which this sprout statement occurs. |
| 35 | RUNME | If set, continue to run the process in which this sprout statement occurs. |

**RESUME OPTIONS**

33-32 READYME If 33-32 is 1, then the current process will not be suspended, but be made ready.

KILLME If 33-32 is 2, then the current process will be terminated.

IRUN If 33-32 is 3, then the current process will not be suspended, but be made running. The newly resumed process will be made ready.

34 This should always be zero.

35 NOTNOW If set, this bit makes the newly resumed process ready instead of running. If 33-32 are not 3, then this bit causes a rescheduling.

**CAUSE OPTIONS**

35 DONTSAVE Never put the <event item> on the notice queue. If there is no process on the wait queue, this makes the cause statement a no-op.

34 TELLALL Wake all processes waiting for this event. Give them all this item. The highest priority process will be made running, others will be made ready.

33 RESCHEDULE Reschedule as soon as possible (i.e., immediately after the cause procedure has completed executed).

**INTERROGATE OPTIONS**

35 RETAIN Leave the event notice on the notice queue, but still return the notice as the value of the interrogate. If the process goes into a wait state as a result of this Interrogate, and is subsequently awakened by a Cause statement, then the DONTSAVE bit in the Cause statement will over ride the RETAIN bit in the Interrogate if both are on.

34 WAIT If the notice queue is empty, then suspend the process executing the interrogate and put its process item on the wait queue.

33 RESCHEDULE Reschedule as soon as possible (i.e., immediately after execution of the interrogate procedure).

32 SAY_WHICH Creates the association EVENT_TYPE ⊗ <event notice> ≡ <event type> where <event type> is the type of the event returned. Useful with the set form of the Interrogate construct.

# APPENDIX F

## Statement Counter System

### GENERAL DISCUSSION

The statement counter system allows you to determine the number of times each statement in your program was executed. Sail accomplishes this by inserting an array of counters and placing AOS instructions at various points in the object program (such as in loops and conditional statements). Sail automatically calls K_ZERO to zero the counter array before your program is entered and K_OUT to write the array before exiting to the system. If your program does not exit by falling out the bottom, or you are interested only in counts during specific periods, then you may declare K_OUT and K_ZERO as external procedures and call them yourself.

Another program, called PROFIL, is used to merge the listing file produced by the Sail compiler with the file of counters produced by the execution of your program. The output of the PROFIL program is an indented listing with execution counts in the right hand margin.

Since the AOS instructions access fixed locations, and they are placed only where needed to determine program flow, they should not add much overhead to the execution time. Although no large study has been made, the counters seem to contribute about 2% to the execution time of the profile program, which has a fairly deeply nested structure.

### HOW TO GET COUNTERS

In order to use the counter system you must generate a listing and also specify the /K switch. Specifying /K automatically selects /10F, since the PROFIL program needs this listing format. The characters '002 and '003 in the listing mark the location of counters.

At the end of each program (i.e. each separate compilation) is the block of counters, preceded by a small data block used by K_ZERO and K_OUT. This block contains the number of counters, the name of the list file, and a link to other such blocks. The first counter location is given the symbolic name .KOUNT, which is accessible from DDT, but cannot be referenced by the Sail program itself.

K_OUT uses GETCHAN to find a spare channel, does a single dump mode output which writes out all the counters for all the programs loaded having counters, and then releases the channel. The file which it writes is xxx.KNT, where xxx is the name of the list file of the first program loaded having counters (usually the name of the Sail source file). If there are no counters, K_OUT simply returns.

### PROFILE PROGRAM

The program PROFIL is used to produce the program profile, i.e. the listing complete with statement counts. It operates in the following manner. First it reads in the file xxx.KNT created by the execution of the user program. This file contains the values of the counters and the names of the list files of the programs loaded which had counters. It then reads the the list files and produces the profile.

The format of the listing is such that only statements executed the same number of times are listed on a single line. In the case of conditional statements, the statement is continued on a new line after the word THEN. Conditional expressions and case expression, on the other hand, are still listed on a single line. In order that you might know the execution counts, they are inserted into the text surrounded by two "brokets" (e.g. <<15>>).

PROFIL expects a command string of the form

<output>←<input> {switches}

where <input> is the name of the file containing the counters; extension .KNT is assumed. If the output device is the DSK, the output file will have a default extension of .PFL. Although the line spacing will probably be different from the source, PROFIL makes an effort to keep any page spacing that was in the source. The switches allowed by PROFIL are

| | |
|---|---|
| /nB | Indent n spaces for blocks (default 4) |
| /nC | Indent n spaces for continuations (default 2) |
| /F | Fill out every 4th line with "..." (default ON) |
| /I | Ignore comments, strip them from the listing |
| /nK | Make counter array of size n (default 200) |
| /nL | Maximum line length of n (default 120) |
| /N | Suppress /F feature |
| /S | Stop after this profile |
| /T | TTY mode = /1C/2B/F/80L |

SAMPLE RUN

Suppose that you have a Sail program named FOO.SAI for which you desire a profile. The following statements will give you one.

```
.EX /LIST FOO(K)  (or TRY or DEB or what have you)
. . . any input to FOO . . .

EXIT

↑C
.R PROFIL
*FOO←FOO/T/S

EXIT

↑C
```

At this point, the file FOO.PFL contains the profile, suitable for typing on the TTY or editing.

## APPENDIX G

## Array Implementation

Let STRINGAR be 1 (TRUE) if the array in question is a String array, 0 (FALSE) otherwise. Then a Sail array of n dimensions has the following format:

```
HEAD:   →DATAWD      ;→ MEANS "POINTS AT"
        HEAD-END-1
ARRHED: BASE_WORD    ;SEE BELOW
        LOWER_BD(n)
        UPPER_BD(n)
        MULT(n)
        ...
        LOWER_BD(1)
        UPPER_BD(1)
        MULT(1)
        NUM_DIMS,,TOTAL_SIZE
DATAWD: BLOCK TOTAL_SIZE
        <sometimes a few extra words>
END:    400000,,→HEAD
```

HEAD — The first two words of each array, and the last, are control words for the dynamic storage allocator. These words are always present for an array. The array access code does not refer to them.

ARRHED — Each array is preceded by a block of 3*n+2 control words. The BASE_WORD entry is explained later.

NUM_DIMS — This is the dimensionality of the array. If STRINGAR, this value is negated before storage in the left half.

DATAWD — This is stored in the core location bearing the name of the array (see symbols, page 141). If it is a string array, DATAWD+1 is stored instead.

TOTAL_SIZE — The total number of accessible elements (double if STRINGAR) in the array.

BOUNDS — The lower bound and upper bound

for each dimension are stored in this table, the left-hand index values occupying the higher addresses (closest to the array data). If they are constants, the compiler will remember them too and try for better code (i.e. immediate operands).

MULT — This number, for dimension m, is the product of the total number of elements of dimensions m+1 through n. MULT for the last dimension is always 1.

BASE_WORD — This is DATAWD minus the sum of (STRINGAR+1) * LOWER_BD(m) * MULT(m) for all m from 1 to n. If this is a string array then the left half is -1.

The formula for calculating the address of A[I,J,K] is:

```
address(A[I,J,K]) =
   address(DATAWD) +
    (I-LOWER_BD(1))*MULT(1) +
    (J-LOWER_BD(2))*MULT(2) +
    (K-LOWER_BD(3))
```

This expands to

```
address(A[I,J,K]) =
  address(DATAWD) +
    I*MULT(1) + J*MULT(2) + K
     -(LOWER_BD(1)*MULT(1) +
        LOWER_BD(2)*MULT(2) +
          LOWER_BD(3)
```

which is

```
BASE_WORD + I*MULT(1) + J*MULT(2) + K
```

By pre-calculating the effects of the lower bounds, several instructions are saved for each array reference.

The LOADER gets confused if BASE_WORD does not designate the same segment as DATAWD. The difference between BASE_WORD and the address of any location in the array should be less than '400000. Avoid constructs like INTEGER ARRAY X[1000000:1000005]. Declare large static arrays last.

# APPENDIX H

## String Implementation

### STRING DESCRIPTORS

A Sail String has two distinct parts: the descriptor and the text. The descriptor is unique and has the following format:

```
WORD1: CONST,,LENGTH
WORD2: BYTP
```

1) CONST. This entry is 0 if the String is a constant (the descriptor will not be altered, and the String text is not in String space, is therefore not subject to garbage collection), and non-zero otherwise.

2) LENGTH. This number is zero for any null String; otherwise it is the number of text characters.

3) BYTP. If LENGTH is 0, this byte pointer is never checked (it need not even be a valid byte pointer. Otherwise, an ILDB machine instruction pointed at the BYTP word will retrieve the first text character of the String. The text for a String may begin at any point in a word. The characters are stored as LENGTH contiguous characters.

A Sail String variable contains the two word descriptor for that variable. The identifier naming it points to WORD1 of that descriptor. If a String is declared INTERNAL, a symbol is formed to reference WORD2 by taking all characters from the original name (up to 5) and concatenating a "." (OUTSTRING's second word would be labeled OUTST.).

When a String is passed by reference to a procedure, the address of WORD2 is placed in the P-stack (see page 160). For VALUE Strings both descriptor words are pushed onto the SP stack.

A String array is a block of 2-word String descriptors. The array descriptor (see page 157) points at the second word of the first descriptor in the array.

Information is generated by the compiler to

allow the locations of all non-constant strings to be found for purposes of garbage-collection and initialization. All String variables and non-preloaded arrays are cleared to NULL whenever a Sail program is started or restarted. The non-constant strings in Preloaded arrays are also set to null by a restart.

INEXHAUSTIBLE STRING SPACE

The string garbage collector expands string space (using discontiguous blocks) whenever necessary to satisfy the demand for places to put strings.

Here are some points of interest:

1) The initial string space size is settable via REQUIRE or the ALLOC sequence. Each string-space increment will be the same as the original size. The threshold (see below) for expansion is 1/8 the string space size (increment size). One can modify these values with USERCON or by storing directly into GOGTAB.

```
 NAME              VALUE
STINCR    LH: # chars in increment
          RH: 4+ # words in increment

STREQD    LH: # chars in threshold
          RH: # words in threshold
```

2) (the threshold) Assume that the garbage collector was called to make room for R characters, and that after garbage collection M-1 discontiguous string spaces are full, with the M'th having N free characters. If N is less than or equal to R+LH (STREQD) then expansion to M+1 string spaces takes place. In other words, if STREQD is 1/8 the size of the current space then that space will not be allowed to become more than about 7/8 full. All but the current space are allowed to become as full as possible, however.

3) Wizards may cause the garbage collector to keep some statistics by setting SGCTIME to -1.

# APPENDIX I

## Save/Continue

A (new) save/continue facility has been implemented in the Sail compiler. This allows compiling header files, saving the state of the compiler, and resuming compilation at a later time. The save/continue facility works with files as the basic unit; compilation can be interrupted only at the end of a file. The /X (eXtend) switch controls the new feature. The examples shown here are for TOPS-10. Analogous commands work under TENEX, using the TENEX RUN and SAVE commands. Example:

```
.R SAIL
*INTRMD.REL[PRJ,PRG]←A,B,C/X
 A.SAI 1 etc.

SAVE ME FOR USE AS XSAIL
EXIT
.SAVE XSAIL
JOB SAVED IN 25K
UPPER NOT SAVED!

.RU XSAIL
*FINAL←D,E,F
 D.SAI
Copying DSK:INTRMD.REL[PRJ,PRG]
 2 3 etc.

*↑C
```

The above is equivalent to

```
.R SAIL
*FINAL←A,B,C,D,E,F
```

On TENEX, the user will want to save all of core when creating the XSAIL.SAV file.

Information is saved in XSAIL.SAV and in the binary file from the first "compilation" (in this case INTRMD.REL). When compilation is resumed, the final binary file is initialized by copying the intermediate file. Save/continue is not allowed if the file break occurs while scanning false conditional compilation or actual parameters to a macro call.

A hint on using this feature: If the source

term of your command string consists of just one file, and this one file does REQUIREs of other source files, the following setup works well.

```
Original file FOO.SAI:
      BEGIN "FOO"
        REQUIRE "[][]" DELIMITERS;
        DEFINE !=[COMMENT];
        REQUIRE "BAZ.SAI" SOURCE_FILE;
        REQUIRE "MUMBLE.SAI" SOURCE_FILE;
        :
        <rest of file>
        :
      END "FOO"

New file FOO.SAI:
   IFCR NOT DECLARATION(GARPLY) THENC
      BEGIN "FOO"
        REQUIRE "[][]" DELIMITERS;
   DEFINE GARPLY=TRUE;
        DEFINE !=[COMMENT];
        REQUIRE "BAZ.SAI" SOURCE_FILE;
        REQUIRE "MUMBLE.SAI" SOURCE_FILE;
   ENDC;
        :
        <rest of file>
        :
      END "FOO"

New file FOO.HDR:
   IFCR NOT DECLARATION(GARPLY) THENC
      BEGIN "FOO"
        REQUIRE "[][]" DELIMITERS;
   DEFINE GARPLY=TRUE;
        DEFINE !=[COMMENT];
        REQUIRE "BAZ.SAI" SOURCE_FILE;
        REQUIRE "MUMBLE.SAI" SOURCE_FILE;
   ENDC;


Initial compilation:
   .R SAIL
   *FOO.INT[PRJ,PRG]←FOO.HDR/X

   SAVE ME!
   .SAV XSAIL
```

Now the command string

```
   FOO←FOO
```

will work both in the case of .R SAIL and in the case .RU XSAIL.

## APPENDIX J

### Procedure Implementation

When a procedure is entered it places three words of control information on the run time (P) stack. This "mark stack control packet" contains pointers to the control packets for the procedure's dynamic and static parents. Register F ('12) is set to point at this area. This pointer is then used to access procedure parameters and other "in stack" objects, such as the local variables of a recursive procedure. Many of the run-time routines (including the string garbage collector) use rF to find vital information. Therefore, THE USER MUST NOT HARM REGISTER '12. If you wish to refer in assembly language to a procedure parameter, the safest way is name it, and let Sail do the address arithmetic. (Similarly one may use the ACCESS construct).

STACK FRAME

Shown here is the stack frame of a recursive procedure.

```
      :.............................:
      :     parameter  1            :
      :.............................:
      :                             :
      :.............................:
      :     parameter  n            :
      :.............................:
      :              :   ret. addr  :
      :.............................:
rF →  :              : dynamic link : (old rF)
      :.............................:
      : →proc desc   : static link  : (rF of static
      :.............................:     parent)
      :  old value of rSP           :
      :.............................:
      : start of recursive locals   :
      :.............................:
      :                             :
      :.............................:
rP →  : end of recursive locals     :←(rP points
      :.............................:  here after
      : start of working storage    :  entry to a
      :.............................:  recursive
      :                             :  procedure)
      :.............................:
```

If a formal parameter is a value parameter then

the actual parameter value is kept on the stack. If a formal parameter is a reference parameter, then the address of the actual parameter is put on the stack. Non-own string locals (to recursive procedures) and string value parameters are kept on the string (SP = '16) stack. The stack frame for a non-recursive procedure is the same except that there are no local variables on the stack. The stack frame for a SIMPLE procedure consists only of the parameters and the return address.

ACCESSING THINGS ON THE STACK

SIMPLE procedures access their parameters relative to the top-of-stack pointers SP(for strings) and P (for everything else). Thus the the k'th (of n) string value parameter would be accessed by

```
OP      AC,2*k-2*n(SP)   ;(SP='16)
```

and the j'th (of m) "arithmetic" -- i.e., not value string -- parameter would be accessed by

```
OP      AC,j-m-1(P)      ;(P='17)
```

Non-SIMPLE procedures use rF (register '12) as a base for addressing parameters and recursive locals. Thus the j'th parameter would be accessed by

```
OP      AC,j-m-2(rF)
```

or, in the case of a string, by

```
MOVE    ACX,2(rF)        ;points at top of
                         ;string stack when
                         ;proc was entered
OP      ACY,2*k-2*m(ACX)
```

Similarly, recursive locals are addressed using positive displacements from rF.

An up-level reference to a procedure's parent is made by executing the instruction

```
HRRZ    AC,1(rF)   ;now AC points at
                   ;stack frame of parent
```

and then using AC in the place of rF in the access sequences above, iterating the process if need be to get at one's grandparent, or some more distant lexical ancestor.

NOTE: When Sail compiled code needs to make such an up-level reference it keeps track of any intermediate registers (called "display" registers) that may have been loaded. Thus, if you use several up-level references together, you only pay once for setting up the "display", unless some intervening procedure call or the like should cause Sail to forget whatever was in its accumulators. Note here that if a display register is thrown away, there is no attempt to save its value. At some future date this may be done. It was felt, however, that the minimal (usually zero) gain in speed was just not worth the extra hair that this would entail.

ACTIONS IN THE PROLOGUE FOR NON-SIMPLE PROCEDURES

The algorithm given here is that for a recursive procedure being declared inside another procedure. The examples show how it is simplified when possible.

1. Pick up proc descriptor address.
2. Push old rF onto the stack.
3. Calculate static link. (a). Must loop back through the static links to grab it. (b). once calculated put together with the PDA and put it on the stack.
4. Push current rSP onto the stack.
5. Increment stack past locals & check for overflow.
6. Zero out whatever you have to.
7. Set rF to point at the MSCP.

EXAMPLES:

1. A non-recursive entry (note: in this section only cases where F is needed are considered).

```
PUSH    P,rF             ;SAVE DYNAMIC LINK
SKIPA   AC,rF
MOVE    AC,1(AC)         ;GO UP STATIC LINK
HLRZ    TEMP,1(AC)       ;LOOK AT PDA IN STACK
CAIE    TEMP,PPDA        ;IS IT THE SAME AS PARENTS
JRST    .-3              ;NO
HRLI    AC,PDA           ;PICK UP PROC DESC
PUSH    P,AC             ;SAVE STATIC LINK
PUSH    P,SP
HRRZI   rF,-2(P)         ;NEW RF
```

In the case that the procedure is declared in

the outer block we don't need to worry about the static link and the prologue can look like

```
PUSH    P,rF            ;SAVE DYNAMIC LINK
PUSH    P,[XWD PDA,0]   ;STATIC LINK WORD
PUSH    P,SP            ;SAVE STRING STACK
HRRZI   rF,-2(P)        ;NEW F REGISTER
```

2. Recursive entry -- i.e one with locals in the stack.

```
PUSH    P,rF            ;SAVE DYNAMIC LINK
SKIPA   AC,rF
MOVE    AC,1(AC)        ;GO UP STATIC LINK
HLRZ    TEMP,(AC)       ;LOOK AT PDA IN STACK
CAIE    TEMP,PPDA       ;IS IT THE SAME AS PARENTS
JRST    .-3             ;NO
HRLI    AC,PDA          ;PICK UP PROC DESC
PUSH    P,AC            ;SAVE STATIC LINK
PUSH    P,SP
HRLZI   TEMP,1(P)       ;
HRRI    TEMP,2(P)       ;
ADD     P,[XWD  locals, locals] ;create space for
CAIL    P,0                     ;arith locals
<trigger pdl ov error>
SETZM   -1(TEMP)        ;zero out locals
BLT     TEMP,(P)        ;
HRLZI   TEMP,1(SP)
HRRI    TEMP,2(SP)
ADD     SP,[XWD 2* string locals,2* string locals]
CAIL    SP,0            ;check for pdl ov
<cause pdl ov error>
SETZM   -1(TEMP)
BLT     TEMP,(SP)       ;zero out string locals
HRRZI   rF,- locals-3(P)
```

The BLT of zeros is replaced by repeated pushes of zero if there are only a few locals. Again, the loop is replaced by a simple push if the procedure is declared in the outer block.

## ACTIONS AT THE EPILOGUE FOR NON-SIMPLE PROCEDURES

1. If returning a value, set it into 1 or onto right spot in the string stack.
2. Do any deallocations that need to be made.
4. Restore rF.
5. Roll back stack.
6. Return either via POPJ P, or by JRST @mumble(P)

EXAMPLES:

1. No parameters.

```
        <step 1>
        <step 2>
        MOVE    rF,(rF)
        SUB     P,[XWD M+3,M+3] ;M= # LOCAL VARS
        POPJ    P,
```

2. n string parameters, m other parameters, k string locals on stack, j other locals on stack.

```
        <step 1>
        <step 2>
        MOVE    rF,(rF)
        SUB     SP,[XWD 2*k+2*n,2*k+2*n]
        SUB     P,[XWD j+m+3,j+m+3]  ;POP STACK
        JRST    @m+1(P)
```

SIMPLE procedures are similar, except that rF is never changed.

## PROCEDURE DESCRIPTORS

Procedure descriptors are used by the storage allocation system, the interpretive caller, BAIL, and various other parts of Sail. They are not put out for SIMPLE procedures. The entries are shown as they are at the present time. No promise is made that they will not be different tomorrow. If you do not understand this page, do not worry too much about it.

```
-1:   link for pd list
 0:   entry address
 1:   word1 of string for proc name
 2:   word2 of string for proc name
 3:   type info for procedure,,sprout defaults
 4:   # string params*2,,# arith params+1
 5:   < ss displ,, + as displ
 6:   lexic lev,,→→local var info
 7:   display level,,→→proc param stuff
10:   pda,,0
11:   pcnt at end of nlksemt,,parent's pda
12:   pcnt at prdec,,loc for jrst exit
13:   type info for first argument,,0 (or →→default value)
      :
      type info for last argument,,0 (or →→default value)
lvi:  byte (4)type(9)lexical-level(23)location
      :
      :
```

The type codes in the lvi (local variable info) block are as follows:

```
type = 0     end of procedure area
type = 1     arith array
type = 2     string array
type = 3     set or list
type = 4     set or list array
type = 5     foreach search control block
type = 6     list of all processes dependent on
             this block.
type = 7     context
type = 10    a cleanup to be executed
type = 11       record pointer
type = 12    record pointer array
type = 17    block boundary. Location gives base
             location of parents block's information.
```

local variable info for each block is organized as

```
info for var
:
info for var
17,lev,loc of parent block bbw
```

## REFERENCES


BBNEXEC — Bolt Beranek and Newman, TENEX Executive Manual, Cambridge, Massachusetts, April 1973.

Feldman — J.A. Feldman and P.D. Rovner, An Algol-Based Associative Language, CACM 12, 8 (August 1969), 439-449.

J.A. Feldman, J.R. Low, D.C. Swinehart, and R.H. Taylor, Recent Developments in SAIL, AFIPS FJCC 1972, 1193-1202.

Frost — M. Frost, UUO Manual (Second Edition), Stanford Artificial Intelligence Laboratory Operating Note 55.4 (July 1975).

Harvey — B. Harvey (M. Frost, ed.), Monitor Command Manual, Stanford Artificial Intelligence Laboratory Operating Note 54.5 (January 1976).

JSYS — Bolt, Beranek, and Newman, TENEX JSYS Manual, Cambridge, Massachusetts, September 1973.

vanLehn — K. vanLehn, SAIL, SAILON 57.3, (June 1973).

MonCom — [Harvey], [BBNEXEC], [OSCMA]

Nauer — P. Nauer (ed.), Revised Report on the Algorithmic Language ALGOL-60, CACM 6 (1963) 1-17.

OSCMA — decsystem10 Operating System Commands Manual DEC-10-OSCMA-A-D, Digital Equipment Corporation, Maynard, Massachusetts, May 1974.

Petit — P. Petit (R. Finkel, ed.), RAID Manual, SAILON 58.2, (March 1975).

Reiser J.F. Reiser, BAIL--A Debugger for SAIL, Stanford Artificial Intelligence Laboratory Memo AIM-270, Computer Science Department Report STAN-CS-75-523, October 1975.

Savitzky S.R. Savitzky (L. Earnest, ed.) Son of Stopgap, SAILON 50.3, March 1971.

SmithN N. Smith, Sail Tutorial, Stanford Aritifical Intelligence Laboratory Memo AIM-290, Computer Science Department Report STAN-CS-76-575, August 1976.

SmithR R. Smith, TENEX SAIL, Institute for Mathematical Studies in the Social Sciences T.R. 248, Stanford University, January 1975.

Swinehart & Sproull D.C. Swinehart and R.F. Sproull, SAIL, SAILON 57.2, (January 1971).

SysCall [Frost], [JSYS], [TopHand]

TopHand decsystem10 Assembly Language Handbook DEC-10-NRZC-D, Digital Equipment Corporation, Maynard, Massachusetts, 1973.

# SECTION 1

# New Features

This section describes changes and additions to Sail since the August 1976 manual, AIM-289.

## 1.1 - Double Precision

Double precision floating-point arithmetic is available. Use the <type_qualifier> LONG in declarations. For example,

```
LONG REAL X, Y, Z;
LONG REAL ARRAY XA[0:N];
```

Currently LONG has meaning only when it appears as part of LONG REAL. (At some future time LONG INTEGERs may also exist.)

The runtime routines LREALIN and LREALSCAN operate the same as REALIN and REALSCAN, except for returning LONG REAL values. The routine CVEL takes a LONG REAL value and returns a string representation like that of CVE, except that "@@" is used to signify LONG when delimiting the exponent. Any of "@", "@@", "E", or "D" are acceptable exponent delimiters to LREALIN and LREALSCAN.

Variables which are declared LONG REAL are represented in KI10 hardware format double precision, take two consecutive words of storage, and provide 62 bits of precision (approximately 18 decimal digits) to represent the fraction part of a floating-point number. Regular REAL variables occupy a single word and have 27 bits (8 decimal digits) of precision. The exponent range of both REAL and LONG REAL variables is from -128 to 127, where 2↑127 is approximately 10↑38.

LONG REAL is a dominant type in arithmetic operations +-*/%↑ MAX MIN and arithmetic relationals <>=≠≤≥. If one operand is LONG REAL then both operands will be converted to LONG REAL (if necessary) before performing the operation. An exponentiation involving a LONG REAL raised to a positive integer constant is an exception to this rule. The type coercion path is linear: STRING ↔ INTEGER ↔ REAL ↔ LONG REAL. Conversion from REAL to LONG REAL is performed by assigning the (only) word of the REAL to the most significant word of the LONG REAL and setting the second (least significant) word of the LONG REAL to zero. Conversion from LONG REAL to REAL is by UUO which rounds.

Arithmetic and assignment operations are compiled into DFAD, DFSB, DFMP, DFDV, DMOVE, DMOVEM instructions. The Sail operations ASH, LSH, ROT, LAND, LOR, EQV, XOR are

performed on both words (ASHC, LSHC, ROTC, 2 ANDs, 2 IORs, etc.). LOCATION of a LONG REAL variable gives an address E such that DMOVE AC,E fetches the appropriate words of memory. When passed by value as an actual parameter to a procedure, both words are placed on the P stack: PUSH P,X ↔ PUSH P,X+1. LONG REAL fields in record classes are handled much like STRING fields, except that the address in the record field points to the first word of a 2-word block (rather than to the second word as in the case with STRINGs).

LONG REAL ARRAYs are stored as contiguous blocks of 2-word values. ARRTRAN done on two LONG REAL arrays is a transparent operation, but for ARRYIN, ARRYOUT, or ARRBLT the actual word count is specified; think about whether you should multiply by 2! At runtime the array descriptor for a LONG ARRAY has bit 12 (40,,0 bit) set in MULT(n), the multiplier for the last dimension (which would otherwise be =1). Similarly, a LONG ARRAY is allocated by setting bit 12 (40,,0) bit in the parameter which specifies the number of dimensions to ARMAK.

Runtime support for LEAP items with LONG REAL datums does not yet exist, although the compiler does generate suitable code. Runtime support for double precision exponentiation is also limited for the moment. Any exponentiation X↑K where K is a positive integer constant is compiled inline using the binary ("Russian peasant") method, regardless of the type of X. Other exponentiations involving LONG REALs are merely translated into procedure calls on

```
LONG REAL PROCEDURE DPOW (INTEGER EXPONENT; LONG REAL BASE);
LONG REAL PROCEDURE DLOGS (LONG REAL EXPONENT, BASE);
```

depending on the type of the exponent. The Sail runtime system does not yet contain such procedures, so you will have to roll your own.

## 1.2 - Declarations and Scope

Sail declarations must occur before use. For example, in the following program the argument to PRINT is interpreted as the K on line 2, even though by the ALGOL60 notion of scope it should be interpreted as the K on line 5.

```
BEGIN "FOO"
INTEGER K;      COMMENT this is line 2;
  BEGIN "RUB"
  PROCEDURE BAR; BEGIN PRINT(K) END;
  INTEGER K;    COMMENT this is line 5;
  <statements>
  END "RUB"
END "FOO"
```

## 1.3 - Two-character Operators

The compiler now recognizes "**" for "↑", ":=" for "←", "<=" for "≤", and ">=" for "≥".

## 1.4 - Requires

REQUIRE OVERLAP_OK; will suppress the message which occurs at initialization when two programs have declared items.

REQUIRE VERIFY_DATUMS; causes the compiler to generate three additional instructions for each DATUM reference, to make sure (dynamically, at run time) that the type of the item in the DATUM construct is the same as the compiler expected. This is similar to (the unimplimented effect of) declaring all itemvars CHECKED. It is planned that VERIFY_DATUMS will soon be a bit in the /A switch and that the corresponding REQUIRE will disappear.

REQUIRE PROCESSES; insures that MAINPR, the main process, is initialized. You need not specify this REQUIRE if you use APPLY or SPROUT, but if the only use of processes is via INTSET then you must REQUIRE PROCESSES;.

## 1.5 - CASE statement

In an explicitly numbered CASE statement the word ELSE can appear where a bracketed case number is normally used. The statement following the ELSE is a catch-all for any case number not mentioned, including anything which would otherwise generate a CASE index error. For example,

```
CASE K OF BEGIN [3] J←3; ELSE J←4; [5] J←5 END
```

is another way of accomplishing

```
    IF K=3 THEN J←3
ELSE IF K=5 THEN J←5
ELSE J←4
```

A CASE statement containing an ELSE case does not generate a call to the CSERR runtime routine, and in addition the jump table usually contains only max_case - min_case +1 words (rather than max_case +1).

## 1.6 - Circular RECORD_CLASSes

To define two record classes, both of which contain RECORD_POINTER fields refering to the other class, say

```
FORWARD RECORD_CLASS BAR (RECORD_POINTER (ANY_CLASS) Q2);
RECORD_CLASS FOO (RECORD_POINTER (BAR) Q1);
RECORD_CLASS BAR (RECORD_POINTER (FOO) Q2);
```

In general, declare one class to be FORWARD and list its RECORD_POINTER fields as pointers to ANY_CLASS. This breaks the circularity and allows maximum compile-time type checking.

# SECTION 2

## Documentation Errors

This is a list of known bugs in the August 1976 Sail manual, AIM-289.

PAGE DESCRIPTION

abstr. "varaiables" is a misspelling [JFR 10-22-76]

iiiL no period after LEAP (line 6 of paragraph) [LES 10-22-76]

162L "i.e" in the line "2. Recursive entry" [JFR 10-23-76]

1R "Nauer" for "Naur" (also References) [JFR 11-2-76]

22L,26L "disjunct" → "conjunct" [JMC 11-12-76]

31L line -9 "its" → "it's" [JMC 11-12-76]

162R The word PDA+'13 contains something other than indicated. The parameter descriptor words actually start at PDA+'14, but the way to find them is to follow the pointer in the right half of PDA+7. [JFR 12-9-76]

9L Another restriction on SIMPLE procedures: They should not do up level addressing themselves (in addition to point 4.) unless the user really understands what is going on with the stack. It is possible to "screw up" without complaints from the compiler. SIMPLE ought to mean "I know what I am doing, so let me do it.". [JFR/DON 12-xx-76]

56L CRLF="('15 & '12)", not '12 & '15 [JFR 1-15-77]

10R It should be made clear that LET A=B; works even if A is a reserved word. In particular, LET DEFINE=REDEFINE; Also note that B can be any reserved word except COMMENT. [COMMENT ALWAYS means "ignore through the next semicolon".]

4R POLLING_POINTS is not a valid <require_spec> [WFW 1-21-77]

50R In FILEINFO, hidate2 occupies 3 bits [JFR 2-3-77]

152L CHNGDB and FILEINFO are defined everywhere except TENEX. [JFR 2-3-77]

# INDEX


∧ (AND) 26
¬ (NOT) 26
∞ in substrings 28
∞, in list REMOVEs 90
∩ (INTERSECTION) 99
∪ (UNION) 99
∨ (OR) 26
!!GO 146
!!GSTEP 146
!!STEP 146
% (integer or real division) 27
& (CONCATENATION), of strings 27
&, of lists 99
-, of sets 99
/ (real division) 27
<>≤≥≠ (RELATIONS) 26
?, Foreach itemvars 93
?, in Binding Booleans 91
?, Matching procedure formals 95

∩ (intersection) 97

∪ (union) 97

≡ (EQV) 150

_ERRJ_ 140
_ERRP_ 139
_SKIP_ 27, 33, 43, 44, 48, 50, 70, 71, 72, 73, 74, 75, 76, 79, 81, 149

$CLASS 66
$REC$ 66
$RECFN 66
$RECGC 66
$SPCAR 67

ABS 28
ACCESS 30
ACOS 51
ADJSP 134
AFTER 88, 89
algebraic variables 6
<algebraic_expression> 22
ALL 88, 90
allocation of variables and arrays 10
AND 26, 88, 150
ANSWER 112, 126
ANY 99
ANY_CLASS 64
ANY, in Binding Boolean 91
ANY, in Derived Sets 92
ANY, in Erase statement 91
ANY, in Foreach 94
APPLY 115
<apply_construct> 114
ARG_LIST 114
<arg_list_specifier> 114
ARGS 144
Array element designation 128
<array_declaration> 3
<array_list> 3
<array_type> 83
Arrays, allocation 10
Arrays, as parameters 7
Arrays, declaration 6
Arrays, initialization and reinitialization 10
Arrays, outer block 5, 7
Arrays, OWN 6
Arrays, PRELOADed 7
Arrays, SAFE declaration 6
Arrays, storage convention 7
ARRBLT 51
ARRCLR 51
ARRINFO 50
ARRTRAN 51
ARRYIN 41, 69
ARRYOUT 41, 69
ASCII 150
ASH 27
ASIN 51
ASKNTC 113, 126
ASND 71
ASSIGN 114
<assign_statement> 114
ASSIGNC 62
<assignc> 56
assignment expressions 25
Assignment statement, semantics 15
<assignment_expression> 22
<assignment_statement> 14
ASSOC 150
ASSOCIATIONS 86
Associations, ERASE 90
Associations, implementation 87
Associations, introduction 83
Associations, MAKE 90
Associations, searching for 91
associative booleans 100
associative context 93
Associative search 91
Associative search, controling hash 91
associative search, relative speeds 95
associative searches, introduction 83
associative store 83, 86
Associative store, searching 91

<associative_statement> 88
ATAN 51
ATAN2 51
ATI 117
attribute 91
AUXCLR 43
AUXCLV 43

<backtracking_statement> 101
Backtracking, introduction 101
BACKUP 43
BAIL 141
BEFORE 88, 89
BIND 91
Binding Boolean 91, 100
Binding Booleans, general considerations 91
<binding_list> 88
BINDIT 99
BINDIT, in Binding Boolean 92
BINDIT, in Derived Sets 92
BINDIT, in Foreach 95
BINDIT, in Foreaches 93
BINDIT, in Matching Procedures 95
BKJFN 71
Block names 1, 140
<block> 1
Boolean Expression <element> 94
<boolean_expression> 22
Boolean, declaration 6
bound 91
Bracketed Triple item 90
Bracketed Triple Item Retrieval 90
Bracketed Triple Item retrieval 92
Bracketed Triple item retrieval, general considerations 91
Bracketed Triple Items, ERASE 91
BREAK 144
BREAKSET 36
BRKERS 124
BRKMAK 124
BRKOFF 124
BUCKETS 91
BUILT_IN 61
Byte pointers, creation 50

CALL 48, 80
CALLER 108
CALLI 48
CASE expressions 25
CASE statement 18
<case_expression> 22
<case_statement> 14
CASEC 60
CAUSE 110
<cause_statement> 110
CAUSE, <options> 110, 155
CAUSE, user defined procedures for 112
CAUSE1 112, 126
Causing events, introduction 110
CFILE 70, 71
character codes 150
CHARIN 71, 79
CHAROUT 71, 79
CHECK_TYPE 61
CHECKED 85, 89
Checked, formal parameters 86
CHECKED, in associative searches 91
Checked, itemvar procedures 86
Checked, type checking 99
CHFDB 71
CHNCDB 51
CHNIOR 43
CHNIOV 43
CHNTAB 120
CLEANUP 10
<cleanup_declaration> 4
CLKMOD 120
CLOSE 35, 69
CLOSF 70, 71
CLOSIN 35, 69
CLOSO 35, 69
CLRBUF 43
CNDIR 81
CODE 48
<code_block> 29
command line 133
<command_line> 132
Comment 1
COMMENTS 130
compile time expressions 58
COMPILER_BANNER 62
COMPILER_SWITCHES 136
<compound_statement> 1
concatenation of lists 99
<cond_comp_statement> 56
conditional compilation 60
Conditional Statements, ambiguity 16
<conditional_expression> 22
<conditional_statement> 14
CONOK 61
Constants, arithmetic 129
Constants, octal 129
Constants, real 129
Constants, string 130
constructive item expressions 98
CONTEXT 101
Context elements 102
<context_declaration> 101
<context_element> 101
CONTINUE statement 19
Conversions, algebraic 23
COORD 144

COP 98, 125
coroutining with RESUMEs 108
COS 51
COSD 51
COSH 51
CPRINT 53
CTLOSW 79
CV6STR 47
CVASC 47
CVASTR 47
CVD 46
CVE 47
CVF 47
CVFIL 50
CVG 47
CVI 87, 123
CVIS 100, 124
CVJFN 71
CVLIST 123
CVMS 59, 60
CVN 87, 123
CVO 46
CVOS 46
CVPS 59
CVS 46
CVSET 123
CVSI 100, 124
CVSIX 47
CVSTR 47
CVXSTR 47

DATUM 85, 89, 128
DATUM, type checking 99
DDT 140, 145
deallocation of variables and arrays 10
DECLARATION (a function) 61
<declaration> 3, 83
default parameters 7
DEFINE 56, 57, 59, 61, 145
<define> 56
DEFPRI 105
DEFPSS 105
DEFQNT 105
DEFSSS 105
DEL_PNAME 100, 124
DELETE 88, 90
DELF 71
delimited strings 58
delimited_anything 61
delimited_expr 61
Delimiters 57
DELIMITERS 57
DELIMITERS, NULL 57
Delimiters, null 57
DELNF 72
DEPENDENTS 106
Derived sets 99
Derived Sets, general considerations 91
<derived_set> 97
DEVST 72
DEVTYPE 72
DFCPKT 126
DFR1IN 117
DFRINT 118
DIRST 81
DISABLE 118
DIV 27
DO statement 18
<do_statement> 14
DOC 56
DONE statement 18
DONTSAVE 111, 155
DPB 50
DRYROT 131, 138
DSKIN 72
DSKOP 71
DSKOUT 72
DTI 117
DVCHR 72

EDFILE 49
EIR 120
<element_list> 88
<element> 88
<element>, Foreach 93
ELSE 14, 22
ELSEC 56
ENABLE 118
ENDC 56
ENTER 36, 69
ENTRY specification 12
EQU 47
EQV 27, 150
ERASE 90
ERASE, in a Foreach 95
ERENAME 36
ERMSBF 49
error messages 138
error procedures 139
ERROR_MODES 138
ERSTR 72
EVALDEFINE 62
EVALREDEFINE 62
event notices 110
Event type items, datums of 112
event types 110
<event_statement> 110
EVENT_TYPE 111, 155
Events, introduction 110
EXP 52
EXPR_TYPE 62
<expression> 22

EXTERNAL declaration 4, 13
EXTERNAL procedures 9, 12

FAIL 89, 95, 106
FALSE, definition 129
FILEINFO 50
FIRST 90, 125
fix (convert real to integer) 23
FIXR 24, 134
float (convert integer to real) 24
FLTR 24, 134
FOR (substringer) 23, 27
FOR statement 17
<for_statement> 14
FORC 60
FOREACH 88
Foreach <element>, Boolean Expression 94
Foreach <element>, List membership 93
Foreach <element>, Retrieval Triple 94
Foreach <element>, Set membership 93
Foreach <element>s 93
Foreach itemvars 92
Foreach searches, relative speeds 95
<foreach_statement> 88
FOREACH, execution of 93
FOREACH, general considerations 91
FOREACH, increase speed of 91
FOREACH, main discussion of 92
Foreach, Matching Procedure <element> 95
Foreach, satisfiers 93
FORGET 101, 102
FORLC 60
formal parameters, Leap 86
formals 7
FORTRAN procedures 9, 13, 20
FORTRAN, actual parameters 10
FORWARD declaration 4
FORWARD procedures 8
FROM 88

GDSTS 72
generation of symbols using macros 59
Gensym 59
GEQ 150
GETBREAK 38
GETCHAN 35, 69
GETFORMAT 46
GETPRINT 53
GETSTS 41, 69
GJINF 81
GLOBAL 86
GNJFN 72
Go To Statements, restrictions 16
GO TO, into a Foreach 92
<go_to_statement> 14
GOGTAB 49, 146

GTAD 81
GTFDB 73
GTJFN 73
GTJFNL 73
GTRPW 120
GTSTS 73
GTTYP 78

handler procedures, Record_class 66
HELP 145

IBP 50
<id_list> 3
identifiers 129
IDPB 50
IDTIM 81
IF expressions 24
IF statement 15
<if_statement> 14
IFC 60
IFCR 61
ILDB 50
ILL MEM REF 131
ILLEGAL UUO 131
IN 88, 89, 150
IN_CONTEXT 51
INCHRS 43, 79
INCHRW 43, 79
INCHSL 43, 79
INCHWL 43, 79
INDEXFILE 73
INF 150
INIACS 50
initialization 10
INITIALIZATION 11
inner block 1
INOUT 41, 69
INPUT 39, 69, 79
INSTR 43, 79
INSTRL 43, 79
INSTRS 44, 79
INT..._APR 154
INT..._INX 153
integer constants 129
Integers, range 6
INTER 150
INTERNAL declaration 4, 12
INTERNAL procedures 9
INTERROGATE 111
<interrogate_construct> 110
INTERROGATE, <options> 111, 155
INTERROGATE, set form of 111
INTERROGATE, user defined procedures for 113
Interrupt codes 153
INTIN 42, 69, 79

INTMAP 118
INTPRO 122
INTRPT 107, 121
INTSCAN 42
INTSET 119
INTTBL 119
INTTY 79
IRUN 108, 155
ISTRIPLE 125
ITEM 84
item booleans 100
<item_expression> 97
<item_primary> 97
ITEM_START 86
<item_type> 83
Item, <typed_item_expression> 128
Items & Itemvars, distinction between 85
Items, ANY 99
Items, BINDIT 99
Items, Bracketed Triple 90
Items, creation of 84
Items, Datums of 85
Items, declared 84
Items, DELETE 90
Items, implementation 86
Items, internal & external 85
Items, internal &external 87
Items, introduction 83
Items, NEW 98
Items, Pnames 100
Items, props of 100
Items, scope 84
Items, type checking 99
Items, type of 85
Items, with array datums 85
ITEMVAR 85
<itemvar_type> 83
Itemvars & Items, distinction between 85
Itemvars, CHECKED 85
Itemvars, implementation 87
Itemvars, initialization 86
Itemvars, scope 86
Itemvars, type checking 85, 89
Itemvars, types of 85

JFNS 74
JFNSL 74
JOIN 109

K_OUT 156
K_ZERO 156
KAFIX 24
KIFIX 24, 134
KILLME 108, 155
KPSITIME 121

Label use 5
<label_declaration> 3
Labels, as actual parameters 10
Labels, restrictions 16
LAND 27
LDB 50
leap booleans 100
LEAP_ARRAY 61
<leap_expression> 97
<leap_relational> 97
<leap_statement> 88
Leap, introduction 83
LENGTH 48, 125
LEQ 150
LET 10
letters, legal Sail letters 129
LEVTAB 120
LIBRARY 11
Library, runtime 33
LINOUT 40, 69
LIST 86
list booleans 100
list element designator 128
List element designators 98
list expressions 99
List membership <element> 93
<list_expression> 97
<list_statement> 88
list, sublists 99
Lists, automatic conversion 89
lists, concatenation 99
lists, initialization 99
Lists, PUT 89
Lists,REMOVE 89
LISTX 125
LNOT 27
LOAD_MODULE 11
LOCATION 28
LODED 44
LOG 52
Logical expressions 27
LOOKUP 36, 69
loop block 19
LOP 48, 98, 125
LOR 27
LSH 27

Macro bodies 58
Macro bodies, concatenation in 59
macro body delimiters 57
macro declarations 57
Macro declarations, scope 58
macro parameter delimiters 57
<macro_body> 56
<macro_call> 56
Macros with parameters 59

Macros without parameters 57
MAKE 88, 90
MAKE, in a Foreach 95
Matching Procedures 95
Matching procedures, as processes 106
Matching Procedures, sharing memory 96
MAX 26
MEMORY 28
MESSAGE 62
MESSAGE procedures 86
MIN 26
MKEVTT 110, 123
MOD 27
MTAPE 41, 69
MTOPR 74
MULTIN 113
MYPROC 109

NEEDNEXT 19
NEQ 150
NEW 97, 98
NEW_ITEMS 98
*NEW_PNAME 100, 124*
NEW_RECORD 65
NEXT statement 19
NIL 99
No one to run 107
NOJOY 112
NOMAC 62
NOPOLL 107
NOT 26, 150
NOTCQ 112
notice queue 110
NOTNOW 108, 155
NOW_SAFE 21
NOW_UNSAFE 21
NULL DELIMITERS 57
null delimiters mode 57
NULL_CONTEXT 102
NULL_RECORD 64, 65
NULL, definition 130

object 91
ODTIM 81
OF 18, 22
OFC 56
OPEN 33, 69
OPENF 74
OPENFILE 74
operator precedence 25
OR 26, 150
OUT 40, 69, 79
OUTCHR 44, 79
outer block 1
OUTSTR 44, 79
OVERFLOW 52

OWN 5

Parameters, default values 7
parametric procedures 9
PBIN 79
PBOUT 79
PBTIN 79
PHI 99
PMAP 81
Pnames 100
PNAMES 100
POINT 50
POLL 107
Polling points 107
POLLING_INTERVAL 107
<preload_specification> 3
PRELOADed arrays 7
PRESET_WITH 7
PRINT 53
Printnames of items 100
PRIORITY 105(X)
PRIORITY(X) 154
*PRISET 109*
Procedure body, emptiness 5
Procedure Calls, actual parameters 20
Procedure Calls, semantics 19
<procedure_call> 15
<procedure_declaration> 3, 84
<procedure_head> 4
<procedure_type> 84
Procedures, as actual parameters 20
Procedures, assembly language 13
Procedures, declaration 7
Procedures, defaults in declarations 9
procedures, Leap 86
Procedures, parametric 9
Procedures, restrictions 10
Procedures, restrictions on formal parameters 7
Procedures, separately compiled 12
procedures, user error 139
process item 104
process procedure 104
Process procedures, Matching 106
Process procedures, recursive 106
<process_statement> 104
Processes, control of scheduling 106
processes, creation of 104
Processes, dependency of 105
Processes, inside recursive procedures 105
PROCESSES, introduction 104
Processes, resumption of 108
Processes, sharable memory 106
Processes, status of 104
*Processes, suspension of 108*
Processes, termination of 107

Program name, for DDT 1
PROPS 89, 100, 128, 129
PROTECT_ACS 30
Pseudo-teletype functions 44
PSIDISMS 121
PSIMAP 119
PSIRUNTM 121
PSOUT 79
PSTACK 105(X)
PSTACK(X) 154
PSTATUS 109
PTY... 44
PUT 88, 89

QUANTUM 104(X)
QUANTUM(X) 154
question itemvars 95
QUICK_CODE 29

RAID 140
RAN 52
RCHPTR 75
RDSEG 81
ready 104
READYME 108, 155
real constants 129
REALIN 42, 69, 79
Reals, range 6
REALSCAN 42
RECORD_CLASS 64
RECORD_POINTER 64
RECURSIVE declaration 4
RECURSIVE procedures 8
REDEFINE 58
Reentering programs 137
REF_ITEM 114
<ref_item_construct> 114
REFERENCE 7, 9, 20
Reference items 114
RELBREAK 38
RELD 71
RELEASE 35, 69
REMEMBER 101, 102
REMOVE 88, 89
REMOVE, in Foreach 93
RENAME 36, 69
REPLACE_DELIMITERS 57
REQUIRE 11
REQUIRE - indexed by last word of the require statement 62
<require_specification> 4
REQUIREs, list of 4
RESCHEDULE 111, 155
rescheduling of processes 106
RESERVED 61
Restarting programs 137
RESTORE 101, 102
RESUME 108
RESUME, <options> 155
RESUME, <return item> 108
RETAIN 111, 155
retrieval item expression 99
Retrieval Triple <element> 88, 94
RETURN 28
RETURN statement 18
RFBSZ 75
RFCOC 78
RFMOD 79
RFPTR 75
RGCOFF 66
RLJFN 75
RNAMF 75
ROT 27
RPGSW 137
RTIW 120
RUNME 105, 154
running 104
RUNPRG 82
RUNTM 82
RWDPTR 75

SAFE declaration 4
<safety_statement> 15
SAMEIV 125
satisfier group 93
SAY_WHICH 111, 112, 155
SCAN 40
SCANC 40
SCHEDULE_ON_CLOCK_INTERRUPTS 121
scheduling of processes 106
SCHPTR 75
scope, of variables 5
SDSTS 72
SECOND 90, 125
SEGMENT_FILE 11
SEGMENT_NAME 11
SET 86
set booleans 100
Set expressions 99
Set membership <element> 93
<set_expression> 97
<set_statement> 88
SETBREAK 38
SETC 150
SETCHAN 76
SETCP 112, 126
SETEDIT 78
SETFORMAT 46
SETINPUT 76
SETIP 113, 126
SETLEX 145
SETO 150

SETPL 40, 69
SETPRINT 53
Sets, automatic coercion 89
Sets, Derived Sets 99
Sets, initialization 99
Sets, PUT 89
Sets, REMOVE 89
SETSCOPE 147
SETSTS 41, 69
SFCOC 78
SFMOD 79
SFPTR 75
SHORT 3, 4, 6, 24
SHOW 145
SIMPLE declaration 4
simple expressions 25
SIMPLE procedures 8
<simple_formal_type> 84
<simple_type> 83
SIN 51
SIND 51
SINH 51
SINI 76, 79
SIR 120
SIZEF 76
SNAIL commands 132
SOS representation 150
SOURCE_FILE 11, 62
SPROUT 104
SPROUT DEFAULTS 105
<sprout_default_declaration> 104
SPROUT_DEFAULTS 104
<sprout_statement> 104
SPROUT, <options> 104, 154
SQRT 52
Stanford character set 150
START_CODE 29
START_CODE, calling procedures from 31
<statement> 1
STDBRK 39, 69
STDEV 72
STDIR 81
STEP 14
STEPC 56
STI 79
STIW 120
storage reallocation 137
STPAR 79
String constant, as comment 1
string constants 130
String descriptors 158
STRING_PDL 11
STRING_SPACE 11
String, declaration 6
STRINGSTACK 104(X)
STRINGSTACK(X) 154
STSTS 73
STTYP 78
SUBSR 48
SUBST 48
<substring_spec> 23
Substrings 27
<suc_fail_statement> 89
SUCCEED 89, 95, 106
SUCH THAT 88, 150
SUSPEND 108
suspended 104
SUSPHIM 105, 154
SUSPME 105, 154
SWAP 150
Swap statement 15
<swap_statement> 14
SWDPTR 75
switches, in command lines 134
symbols, automatic generation of 59
<synonym_declaration> 4
SYSTEM_PDL 11

TANH 51
TELLALL 111, 155
TERMINATE 107
terminated 104
TEXT 145
THAT 88
THEN 14, 15, 22
THENC 56
THIRD 90, 125
time sharing with processes 120
TMPIN 42, 69
TMPOUT 42, 69
TO 23, 27
TRACE 145
TRAPS 145
TRIGINI 52
Triple, Binding Boolean 91
<triple> 88
TRIPLES 86
Triples, introduction 83
TRUE, definition 129
TTYIN 44, 79
TTYINL 44, 79
TTYINS 44, 79
TTYUP 44, 79
type checking, itemvars 85
type conversions, algebraic 23
<type_qualifier> 3
typed_item_expression 128
<typed_item_expression> 128
TYPEIT 123

unbound 91
UNBREAK 146

UNDELETE 76
UNION 150
UNSTACK_DELIMITERS 57
UNTIL 14, 18
UNTILC 56
UNTRACE 146
URSCHD 107
USER1 112
USER2 112
USERCON 48
USERERR 49
USETI 42, 69
USETO 42, 70
UUOFIX 24

VALUE 7, 9, 20
value 91
<variable> 128
variables 128
Variables, allocation 10
variables, initialization 10
variables, scope 5
VERSION 11, 12

WAIT 111, 155
wait queue 110
WAITQ 112
WHILE 14
WHILE statement 17
<while_statement> 14
WHILEC 60
WORDIN 40, 70
WORDOUT 41, 70

XOR 27, 150